



Опорный конспект курса

"Тестирование программного обеспечения"

Авторы: Е. Бритова, А. Стадник

2017

Оглавление

УРОК 1.

- [1.1. Направления в тестировании. Куда развиваться?](#)
- [1.2. Введение в тестирование \(основные понятия\) и разработку ПО \(QA/QC\)](#)
- [1.3. Проектная документация](#)
- [1.4. Тестовая документация](#)
- [1.5. Требования](#)
 - [Уровни и типы требований](#)
 - [Техники тестирования требований](#)
- [1.6. Архитектура компьютера.](#)

УРОК 2.

- [2.1. Уровни тестирования.](#)
- [2.2. Юнит тесты](#)
- [2.3. Классификация и Виды тестирования](#)

УРОК 3.

- [3.1. Системы контроля версий.](#)
- [3.2. Сборка и развертывание. \(Build and deploy\)](#)
- [3.3. Непрерывная интеграция \(CI.Continuous Integration\)](#)
- [3.4. Техники тест дизайна](#)

УРОК 4.

- [4.1. Понятие дефект](#)
 - [Жизненный цикл дефектов](#)
 - [Классификация дефектов](#)
- [4.2. JIRA](#)

УРОК 5.

- [5.1. Мобильное тестирование](#)
- [5.2. Роли](#)
- [5.3. Жизненный цикл ПО. Методологии - водопад, V-model, Agile \(Scrum\)](#)
 - [Стадии цикла разработки ПО](#)
 - ["Водопад" или каскадная модель](#)
 - [V-модель \(V-model\) - разработка через тестирование](#)
 - [Итеративная модель \(Iterative model\)](#)
- [5.4. Артефакты](#)

УРОК 6.

- [6.1. Сети. IP адрес и порт](#)
- [6.2. Архитектура клиент-сервер](#)
- [6.3. Консоль \(интерфейс командной строки \)](#)

УРОК 7.

[7.1. HTTP протокол](#)

[Методы HTTP](#)

[Код Состояния](#)

[7.2. Инструменты разработчиков](#)

[7.3. Selenium](#)

[Команды Selenium](#)

[Действия \(actions\)](#)

[Локаторы](#)

[Проверки \(checks\)](#)

[Дополнения](#)

[Полезные ссылки](#)

[Рекомендации на время курса](#)

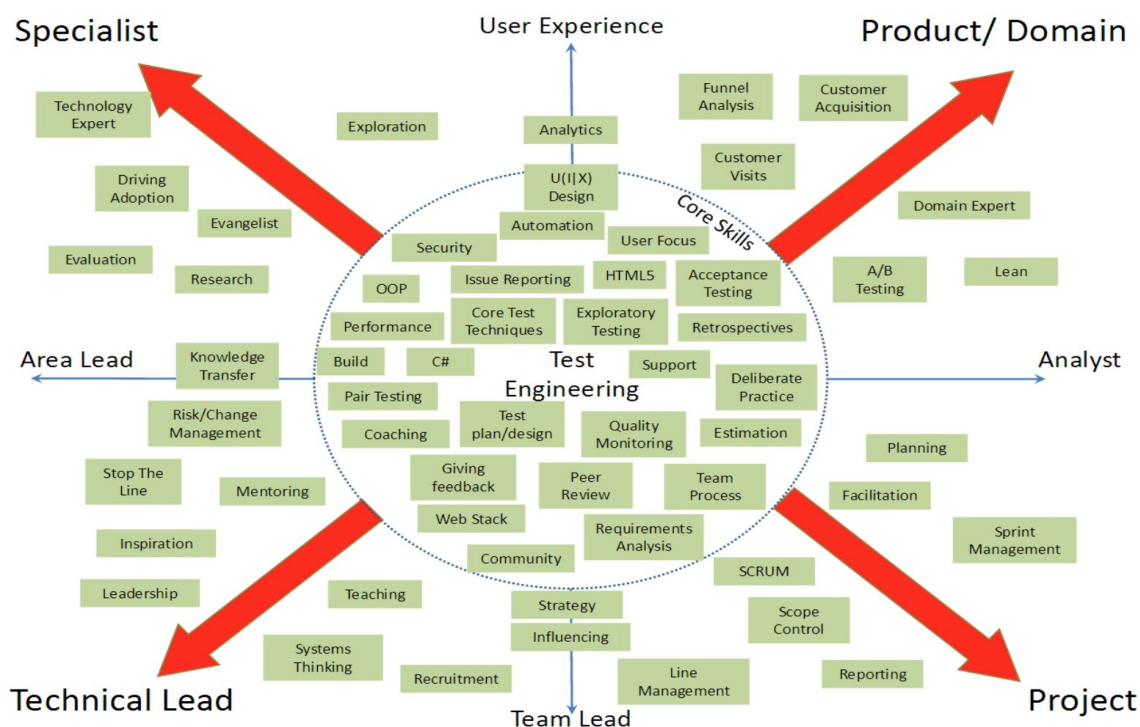
УРОК 1.

1.1. Направления в тестировании. Куда развиваться?

На сегодня QA, наряду с разработкой — ведущее направление в IT-индустрии. Мало кто задумывается о том, что любая программа проходит через тестировщиков. Даже у такого гиганта, как Microsoft, на 1000 строк сырого кода приходится от 10 до 20 ошибок. И только благодаря эффективной работе QA, пользователи получают качественный продукт.

Навыки, необходимые для тестирования - любознательность, внимание к деталям, способность смотреть на общую картину. Новоиспеченный тестировщик должен обладать не только жгучим желанием быть тестировщиком, но и должен приобрести практически с нуля все те навыки, которые необходимо иметь.

Все навыки тестировщика можно собрать в карту. Центр карты представляет собой набор тех основных навыков, в которых каждый тестировщик должен иметь хотя бы базовые знания. От центра расходятся четыре квадранта. Эти квадранты определяют направления, которые в будущем тестировщик может взять за специализацию.



Путем явного просмотра набора основных навыков, таких как исследовательское тестирование, методики испытаний, проблема отчетности, парное тестирование, автоматизация тестирования, намеренная практика, предоставление обратной связи и др., это становится намного проще для тестера для определения того на чем он должен сосредоточиться. **Очень важно**, чтобы основные навыки были изучены, отточены на практике, прежде чем думать о специализированных навыках. Эта схема используется в личных беседах по развитию между тестерами и их руководителем, чтобы помочь найти им сильные и слабые стороны и сформировать цель для дальнейшего обучения.

Рекомендованные статьи:

[Testing: An Obvious Career Choice](#)

[Testing Your Career Path](#)

1.2. Введение в тестирование (основные понятия) и разработку ПО (QA/QC)

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Тестирование - это процесс исследования, испытания программного продукта, имеющий две различные цели:

- продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям и обладает определенным уровнем качества;
- выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Ключевое слово в тестировании да и в разработке ПО в целом - **качество**. Стандарт ISO/IEC 25010:2011 определяет качество программного обеспечения как степень удовлетворения системой заявленных и подразумеваемых потребностей различных

заинтересованных сторон. В соответствии со стандартом модель качества продукта включает восемь характеристик:

- функциональная пригодность;
- уровень производительности;
- совместимость;
- удобство пользования;
- надежность;
- защищённость;
- сопровождаемость;
- переносимость (мобильность).

Состав и содержание документации, сопутствующей процессу тестирования, определяется стандартом IEEE 829-2008 IEEE Standard for Software and System Test Documentation.

Но, к сожалению, даже очень полное и подробное тестирование не может гарантировать отсутствие ошибок в программном продукте.

“Тестирование программ может служить доказательством наличия ошибок, но никогда не докажет их отсутствие!” Эд. Дейкстра, 1972 г.

Невозможно отыскать абсолютно все ошибки в программном продукте. Ошибки остаются всегда. Построение исчерпывающего входного теста невозможно. Иными словами, невозможно полностью протестировать программу: даже для самой простейшей программы это займет большое количество времени, которое никогда не будет восполнено выгодой от производства «идеально оттестированной» программы. Следует стремиться к 100% покрытию тестами, но условия динамичного рынка часто заставляют уменьшать время на разработку и тестирование в пользу уменьшения сроков и стоимости разработки.

Тестирование пронизывает весь жизненный цикл ПО, начиная от проектирования и заканчивая неопределенно долгим этапом эксплуатации. Эти работы напрямую связаны с задачами управления требованиями и изменениями, ведь целью

тестирования является как раз возможность убедиться в соответствии программ заявленным требованиям.

Тестирование — процесс пошаговый, это плановая и упорядоченная деятельность. Этот момент очень важен, поскольку в условиях зачастую очень ограниченного времени, выделенного на разработку и тестирование, хорошо продуманный и систематический подход быстрее приводит к обнаружению программных ошибок, чем плохо спланированное тестирование, проводимое в спешке.

Для повышения продуктивности необходимо придерживаться следующего плана:

- Увеличить вероятность обнаружения багов, выбрав те модули и функции, вероятность появления багов в которых велика;
- Создать тесты, которые могут быть использованы повторно;
- Эффективно использовать ресурсы;
- Вложиться в рамки, предусмотренные графиком и бюджетом проекта;
- Получить на выходе качественный продукт.

При изучении курса часто будут встречаться два понятия **Quality Assurance** (обеспечение качества) и **Quality Control** (Testing) (контроль качества, тестирование). Эти понятия часто путают, но следует помнить, что разница между ними довольно большая, поскольку QC является лишь небольшой частью QA:

Quality Control - это оценка качества продукта в конкретный момент времени, на разных стадиях его разработки, или же другими словами - оценка промежуточных и конечных результатов работы. Это процесс нахождения ошибок в продукте с целью их последующего исправления.

Quality Assurance - это превентивный (предупреждающий) процесс, задачей которого является обеспечение качества продукта в будущем. QA отвечает за качество процессов, которые в свою очередь применяются для создания продуктов.

Таким образом, тестирование, как оценка качества (QC), является лишь частью понятия обеспечения качества (QA).

QA в свою очередь является элементом цепочки Управления Качествен (Quality Management – QM).

Приведем примеры некоторых процессов управления качеством:

- Планирования качества – о качестве продукта, услуг начинают думать с самого начала. Определяются процессы, методологии, стандарты, критерии приемки, контрактные обязательства, целевая среда функционирования продукта и т.д
- Обеспечения качества – отвечает за качество процессов и их применение
- Контроль качества – техники контроля качества артефактов (продуктов)
- Улучшения качества – постоянные, измеримые улучшения процессов с целью сокращения расходов (деньги, время, материалы), оптимизации работы, улучшения качества продукта и т.п

1.3. Проектная документация

Зачем нужна проектная документация в ИТ? Для снижения рисков Заказчика и Исполнителя. **Исполнителю** проектная документация необходима для ведения проекта – управления объемами, сроками, финансами, качеством. **Заказчику** наличие проектной документации поможет оценить качество работ и ощутимо улучшит дальнейшую эксплуатацию решения.

Чаще всего это такие документы:

- **Техническое задание.** Документ, в котором Заказчик описывает что он хочет увидеть в результате. На практике, Исполнитель может и должен принимать участие в разработке ТЗ и помогать Заказчику разобраться в тонкостях продукта, ведь совсем не просто описать то, что еще не используется. Таким образом, работа и согласование ТЗ могут растянуться, но это лучше, чем старт с некорректным ТЗ.
- **Высокоуровневый дизайн** (High-Level Design) описывает предлагаемое решение на уровне архитектуры, не вдаваясь в подробности спецификаций, конфигураций и т.п. Верно будет дать определение HLD как концепту.
- **Низкоуровневый дизайн** (Low-Level Design). Этот документ почему-то вызывает много споров относительно своего содержания, многие говорят о том что конфиги нужно описывать в Implementation Plan, я скажу так – зависит от проекта.

- **Спецификации компонентов.** Это понимается как коммерческое предложение, и совершенно верно. НО не лишним будет приложить datasheets к КП, если у Клиента возникнут вопросы, он может быстро найти на них ответы, что хорошо.
- **План и результаты аудита (Site Survey).** Это вообще-то два “зеркальных” документа, но на практике в Плане вы не сможете учесть 100% объем, поэтому полученные результаты могут изменить изначальный план.
- **План выполнения работ (Project Plan).** Не совсем документ, скорее этап.
- **Тестовая документация** (о ней подробнее далее)
- **Эксплуатационная документация.** Этот документ будет крайне полезен если вы оказываете пост-поддержку проекта, а для сотрудников Заказчика система нова и работа с ней вызывает много однотипных вопросов.

ИТ должно быть единым организмом, и интеграция должна учитываться с начала и до конца проекта. Конкретные вещи по интеграции зависят от конкретной ситуации, и выделить эту тему в отдельный документ не получится в большинстве случаев.

Учитывайте интеграцию проекта с существующими системами везде где только это возможно – этим вы снизите риски и повысите конечную удовлетворенность Заказчика.

1.4. Тестовая документация

Во время проведения тестирования создается и используется определенное количество тестовых артефактов (документы, модели и т.д.). Наиболее распространенными тестовыми артефактами являются: Test Plan, Test Strategy, Check List, Test Case, Test Suite, Test Report, RTM, Test Data.

Тест План / План Тестирования (Test Plan) - это документ, описывающий весь объем работ по тестированию, начиная с описания тестируемых объектов, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Документ должен как минимум отвечать на следующие вопросы:

- *что надо тестировать* (объект тестирования: система, приложение, оборудование)
- *что будете тестировать* (список функций и компонент тестируемой системы)
- *как будете тестировать* (стратегия тестирования – виды тестирования и их применение по отношению к тестируемому объекту)
- *тестовые окружения*, на которых необходимо проверять программный продукт
- *когда будете тестировать* (последовательность проведения работ: подготовка, тестирование, анализ результатов, учёт зависимостей тестовых активностей от задач разработки и смежных групп)

Если в вашей компании сложился **формальный процесс**, в тест-плане также будут уместны:

- перечень согласовывающих лиц
- принятые стандарты и шаблоны
- критерии начала тестирования:
 - готовность тестовой платформы (тестового стенда)
 - законченность разработки требуемого функционала
 - наличие всей необходимой документации
- критерии окончания тестирования: результаты тестирования удовлетворяют критериям качества продукта:
 - требования к количеству открытых багов выполнены
 - выдержка определенного периода без изменения исходного кода приложения Code Freeze (CF)
 - выдержка определенного периода без открытия новых багов Zero Bug Bounce (ZBB)

Рекомендации по написанию Тест Плана: Каждая методология или процесс пытается навязать нам свои форматы оформления планов тестирования. Предлагаю, как пример, шаблоны тест планов от [RUP \(Rational Unified Process\)](#) и стандарт [Test Plan Template IEEE 829](#)

Также рекомендую изучить статью: [Тест-план на одну страницу](#)

Стратегия тестирования (Test Strategy)

Различие задач и целей тестирования на протяжении жизненного цикла продукта приводит к необходимости разрабатывать и реализовывать различные стратегии тестирования. Стратегия тестирования — это план проведения работ по тестированию системы или ее модуля, учитывающий специфику функциональности и зависимости с другими компонентами системы и платформы.

Каждая такая стратегия определяет:

- итерации, на которых используются стратегия тестирования и цели тестирования на каждой итерации;
- стадии тестирования для каждой итерации;
- критерий успешного завершения тестирования;
- типы используемых тестов;
- набор методов и инструментальных средств, необходимых для проведения тестирования и оценки качества;
- критерии оценки тестов.

Стратегии тестирования должны разрабатываться на этапе планирования тестирования.

Чек Лист (Check List) - один из фундаментальных инструментов тестирования. Они позволяют не забывать о важных тестах, фиксировать результаты своей работы и отслеживать статистику о статусе программного продукта. Иногда чек-листами называют подробные инструкции о тестируемом продукте, содержащие последовательность действий, множество деталей и т.д. Это не так! **Главный принцип** чек-листов заключается в том, что каждый тестировщик по-своему проходит их, расширяя тестовый набор своей экспертизой.

Какие **преимущества** чек-листов по сравнению с тест-кейсами:

- нивелирование “эффекта пестицида” в регрессионном тестировании

- расширение тестового покрытия за счет отличий при прохождении
- сокращение затрат на содержание и поддержку тестов: не надо писать много буквочек!
- отсутствие рутины, которую так не любят квалифицированные тестировщики
- возможность проходить и комбинировать тесты по-разному, в зависимости от предпочтений сотрудников

Проверка	Результат		
	Win XP	XP SP4	Win Vista
Операции с файлами	ok	ok	ok
Создание файла	ok	ok	ok
Открытие файла	ok	ok	ok
Сохранение документа	ok	ok	ok
Печать	ok	ok	ok
Редактирование файлов	bugs	bugs	bugs
Отмена	ok	ok	ok
Копирование	ok	ok	ok
Вырезание	bug #146	bug #146	bug #146
Вставка	ok	ok	ok
Удаление	ok	ok	ok
Поиск	bug #123	bug #133	ok
Поиск с заменой	bug #126	ok	ok

При этом, чек-листы сохраняют множество **плюсов**, за которые так популярны детальные тест-кейсы:

- статистика: кто, когда, что проходил (с детализацией по сборке продукта и окружению, на котором проводилось тестирование)
- памятка, которая помогает не забыть важные тесты
- возможность оценить состояние продукта, его готовность к выпуску

Конечно, было бы нечестно рассказать про плюсы и умолчать о **минусах** чек-листов:

- начинающие тестировщики не всегда эффективно проводят тесты без достаточно подробной документации

- чек-листы невозможно использовать для обучения начинающих сотрудников, так как в них недостаточно подробной информации
- заказчику или руководству может быть недостаточно того уровня детализации, который предлагают чек-листы

Рекомендованная ссылка: http://testbase.ru/?post_type=skill&p=55

Test Case

Еще одной обязательной сущностью, с которой столкнется каждый тестировщик, является Test Case (Тестовый случай). Test Case – это тестовый артефакт, суть которого заключается в выполнении некоторого количества действий и/или условий, необходимых для проверки определенной функциональности разрабатываемой программной системы.

Структура данного артефакта заключается в «троице»:

Выполняемое действие (Action) - Ожидаемый результат (Expected result) - Фактический результат (Test result).

Action	Expected Result	Test Result (passed/failed/blocked)
PreConditions		
do A1	verify B1	passed
do A2	verify B2	failed
Test Case Description:		
do A3	verify B3	blocked
PostConditions		

Непосредственно сам тестовый случай состоит из 3 частей:

- Предусловия (PreConditions) – либо список шагов, которые приводят проверяемую систему в состояние, пригодное для тестирования, либо список проверок условий того, что система уже находится в необходимом состоянии.

- Описание тестового случая (Test Case Description) – список действий, с помощью которых осуществляется основная проверка функционала (после которой и сверяется фактический результат с ожидаемым).
- Постусловия (PostConditions) – список действий, которые возвращают систему в исходное состояние.

<i>Description</i>	<i>Steps</i>	<i>Expected results</i>
Что проверяем	Какие шаги надо выполнить	Берется из требований, либо, исходя из здравого смысла

Способ описания тест кейсов и их структура может в каждой компании или команде быть разным: иметь разные глубины описания необходимых действий и результатов, иметь разные структурные составляющие. Но хорошая структурированность и высокая удобность шаблонов тестовых случаев, может весьма сократить время рутинных заполнений форм и повысить эффективность команды в целом.

Тестовый набор (Test Suite) - это комбинация тестовых сценариев, для проверки определенной части программного обеспечения, объединенной общей функциональностью или целями, преследуемыми запуском данного набора.

Test Suite ID		TS-021				
Task Project		mPACT 3.0 (in Requirements Planning)				
Objective		Force preview in Setup				
Modules Covered		mPACT Pro				
Environment		Win 8, Firefox, IE 9, Safari, Google Chrome				
Test Suite History						
Date	Version	Description	Changed by			
03.06.2014	1.1	Test suite created	Inna Sukhenko			
04.06.2014	1.2	Test suite reviewed and updated	Sergey Levchenko			
05.06.2014	1.3	Test cases 03-04 updated	Inna Sukhenko			
Req ID	ID	Priority	Description	Steps	Expected results	Notes
	TC-01	Major	Verify Preview button on Add mode on Area of Interest section	1. Open Setup and click + for adding new TAB 2. Check Area of Interest section for Basic TAB 3. Check Area of Interest section for Advanced TAB 4. Check Area of Interest section for Suggestion TAB 5. Add one keyword 6. Delete last keyword	On steps2,3,4 Preview button should be shown to the left from to Save link and it should be disabled. Save button should be disabled. On step5, Preview button should be enabled and clickable. "Save" button should be disabled. On step6, Preview and Save buttons should be disabled.	
	TC-02	Major	Verify Preview button on Add mode on Competitors section	1. Open Setup for exist view 2. Click Add link on Company and Competitors section 3. Check Company and Competitors section for Basic TAB 4. Check Company and Competitors section for Advanced TAB 5. Check Company and Competitors section for Suggestion TAB 6. Add one keyword 7. Delete last keyword	On steps3,4,5 Preview button should be shown to the left from to Save button and it should be disabled. Save button should be disabled. On step6, Preview link should be clickable. Save button should be disabled. Cancel button should be enabled. On step7, Preview and Save buttons should be disabled. Cancel button should be enabled.	
	TC-03	Minor	Verify Preview button on Edit mode on Area of Interest section	1. Open Setup for exist view 2. Click Edit link on Area of Interest section 3. Check Area of Interest section for Basic TAB 4. Check Area of Interest section for Advanced TAB 5. Check Area of Interest section for Suggestion TAB 6. Delete last keyword 7. Add one keyword	On steps3,4,5 Preview button should be shown to the left from to Save button. Preview, Save and Cancel buttons should be enabled and clickable. On step6, Preview button should be disabled. Save and Cancel buttons should be enabled and clickable. On step7, Preview, Save and Cancel buttons should be enabled and clickable.	

TestReport представляет собой суммарную информацию о прохождении тестов, на основе анализа которых и сравнения с ожидаемыми результатами выполняется детальная оценка качества тестируемого продукта и текущего статуса процесса тестирования. Рекомендуется записывать и сохранять результаты тестирования для каждого этапа как один из важнейших артефактов тестирования.

Структура документа:

<i>Feature to be tested</i>	<i>Status</i>	<i>Assign</i>	<i>Comment</i>
1	Pass		
2	Fail		
3	Blocked		Bug#2
4	In progress		

История таких репортов дает нам статистику уязвимых мест.

RTM (Requirements Traceability Matrix) - это двумерная таблица, содержащая соответствие функциональных требований (functional requirements) продукта и подготовленных тестовых сценариев (test cases). В заголовках колонок таблицы расположены требования, а в заголовках строк - тестовые сценарии. На пересечении - отметка, означающая, что требование текущей колонки покрыто тестовым сценарием текущей строки.

	Feature1	Feature2	Feature3
TestCase1			
TestCase2			
TestCase3			

Расчет тестового покрытия относительно требований проводится по формуле:

$$Tcov = (Lcov/Ltotal) * 100\%$$

где: Tcov - тестовое покрытие

Lcov – количество требований, проверяемых тест кейсами

Ltotal - общее количество требований

Для измерения покрытия требований, необходимо проанализировать требования к продукту и разбить их на пункты. Опционально каждый пункт связывается с тест кейсами, проверяющими его. Совокупность этих связей и является матрицей трассировки. Проследив связи, можно понять какие именно требования проверяет тестовый случай.

Тесты не связанные с требованиями не имеют смысла. Требования, не связанные с тестами - это "белые пятна", т.е. выполнив все созданные тест кейсы, нельзя дать ответ реализовано данное требование в продукте или нет.

TestData (Примеры входных данных). Призваны определять наборы (обычно формальных) входных данных для тестов, для них также возможны ожидаемые результаты или признак – позитивные данные или негативные. Тестовые данные должны храниться в одном месте, желательно в центральном хранилище данных. Очень рекомендуется собирать вместе данные для каждой определенной группы тестов.

Test Data for Login form				
ID	input data for "login"	input data for "password"	input data for "email"	result
TD-002-1	pechkin	pechkin8	pavel.pechkin@gmail.com	active user / pass
TD-002-2	pechkin2	pechkin28	pavel.pechkin2@gmail.com	active user / pass
TD-002-3	pechkin3	pechkin38	pavel.pechkin3@gmail.com	active user / pass
TD-003-1	pechkin0	pechkin0	pavel.pechkin0@gmail.com	not exist user / fail
TD-003-2	pechkin00	pechkin00	pavel.pechkin00@gmail.com	not exist user / fail
TD-004-1	pechkin_old	pechkin_old	pavel.pechkin_old@gmail.com	expired user / fail
TD-004-2	pechkin_old2	pechkin_old2	pavel.pechkin_old2@gmail.com	expired user / fail

Шаблоны документов: <http://www.protesting.ru/testing/templates.html>

1.5. Требования

Начнем с азов: каждый проект преследует свою цель. В долгосрочной перспективе все коммерческие проекты создаются для заработка денег, но в краткосрочной перспективе цели отличаются. Для тиражного продукта целью может быть захват рынка, вывод продукта требуемого качества или просто «релиз до рождественских каникул в США». Для заказного это может быть эффективная презентация прототипа

заказчику или успешное внедрение.

Пример требования на «салфетке»



В этом контексте можно сказать, что: **требования** – описание того, какие функции и с соблюдением каких условий должно выполнять приложение в процессе решения полезной для пользователя задачи.

Более в широком смысле можно сказать, что **требования** (англ. *Requirements*) — совокупность утверждений относительно атрибутов, свойств или качеств программной системы, подлежащей реализации.

Исходя из выявленных проектных требований строится весь последующий процесс: планирование, проектирование, разработка, тестирование. Требования – это фундамент для построения успешного продукта.

Если требования изначально не соответствуют цели, не выявлены, не согласованы между участниками проекта, то даже самая слаженная работа проектной команды не приведёт проект к успеху. При некорректных требованиях вы можете долго и упорно тестировать продукт, находить дефекты, проявлять активность, что-то заводить, исправлять, проверять... Но проекту это пользы не принесёт!

Если поискать определения требований в литературе 10-20-30-летней давности, то можно заметить, что изначально о пользователях, их задачах и полезных для них свойствах приложения в определении требования не было сказано. Пользователь

выступал как абстрактная фигура, не имеющая отношения к приложению. В настоящее время такой подход недопустим, т.к. он не только приводит к коммерческому провалу продукта на рынке, но и многократно повышает затраты на разработку и тестирование.

Уровни и типы требований

Бизнес-требования (*business requirements*) — определяют назначение ПО, описываются в документе о видении (*vision*) и границах проекта (*scope*).

Пользовательские требования (*user requirements*) — определяют набор пользовательских задач, которые должна решать программа, а также способы (сценарии) их решения в системе. Пользовательские требования могут выражаться в виде фраз утверждений, в виде сценариев использования (англ. *use case*), пользовательских историй (англ. *user stories*), сценариев взаимодействия (англ. *scenario*).

Функциональные требования (*functional requirements*) — охватывают предполагаемое поведение системы, определяя действия, которые система способна выполнять. Описывается в системной спецификации (англ. *system requirement specification, SRS*).

Нефункциональные требования (*non-functional requirements*) — описывают свойства системы (удобство использования, безопасность, надежность, расширяемость, быстродействие и т.д.), которыми она должна обладать при реализации своего поведения.

Требования к требованиям

Характеристики качества требований по-разному определены различными источниками. Однако, следующие характеристики являются общепризнанными:

Характеристика	Объяснение
Единичность	Требование описывает одну и только одну вещь.

Завершённость	Требование полностью определено в одном месте и вся необходимая информация присутствует.
Последовательность	Требование не противоречит другим требованиям и полностью соответствует внешней документации.
Атомарность	Требование «атомарно». То есть оно не может быть разбито на ряд более детальных требований без потери завершённости.
Отслеживаемость	Требование полностью или частично соответствует деловым нуждам как заявлено заинтересованными лицами и документировано.
Актуальность	Требование не стало устаревшим с течением времени.
Выполнимость	Требование может быть реализовано в пределах проекта.
Недвусмысленность	Требование кратко определено без обращения к техническому жаргону, акронимам и другим скрытым формулировкам. Оно выражает объективные факты, не субъективные мнения. Возможна одна и только одна интерпретация. Определение не содержит нечётких фраз. Использование отрицательных утверждений и составных утверждений запрещено.
Обязательность	Требование представляет определённую заинтересованным лицом характеристику, отсутствие которой приведёт к неполноценности решения, которая не может быть проигнорирована. Необязательное требование — противоречие самому понятию требования.
Проверяемость	Реализованность требования может быть определена через один из четырёх возможных методов: осмотр, демонстрация, тест или анализ.

Техники тестирования требований

Тестирование документации и требований относится к разряду нефункционального тестирования (*non-functional testing*). Основные техники такого тестирования в контексте требований таковы:

Взаимный просмотр (*peer review*) является одной из наиболее активно используемых техник тестирования требований и может быть представлен в одной из трех следующих форм (по мере нарастания его сложности и цены):

- **Беглый просмотр** (*walkthrough*) — это ситуация, когда вы в школе с одноклассниками проверяли перед сдачей сочинения друг друга, чтобы найти опiski и ошибки.
- **Технический просмотр** (*technical review*) — это ситуация, когда некий договор визирует юридический отдел, бухгалтерия и т.д.
- **Формальная инспекция** (*inspection*) — это ситуация генеральной уборки квартиры (включая содержимое всех шкафов, холодильника, кладовки и т.д.)

Вопросы. Следующей очевидной техникой тестирования и повышения качества требований является (повторное) использование техник выявления требований, а также (как отдельный вид деятельности) — задавание вопросов. Если хоть что-то в требованиях вызывает у вас непонимание или подозрение — задавайте вопросы. Можно спросить представителей заказчика, можно обратиться к справочной информации. По многим вопросам можно обратиться к более опытным коллегам при условии, что у них имеется соответствующая информация, ранее полученная от заказчика. Главное, чтобы ваш вопрос был сформулирован таким образом, чтобы полученный ответ позволил улучшить требования.

Тест-кейсы и чек-листы. Мы помним, что хорошее требование является проверяемым, а значит, должны существовать объективные способы определения того, верно ли реализовано требование. Продумывание чек-листов или даже полноценных тест-кейсов в процессе анализа требований позволяет нам определить, насколько требование проверяемо. Если вы можете быстро придумать несколько пунктов чек-листа, это ещё не признак того, что с требованием всё хорошо (например,

оно может противоречить каким-то другим требованиям). Но если никаких идей по тестированию требования в голову не приходит — это тревожный знак.

Рисунки (графическое представление). Чтобы увидеть общую картину требований целиком, очень удобно использовать рисунки, схемы, диаграммы, интеллект-карты и т.д. Графическое представление удобно одновременно своей наглядностью и краткостью (например, UML-схема базы данных, занимающая один экран, может быть описана несколькими десятками страниц текста). На рисунке очень легко заметить, что какие-то элементы «не стыкуются», что где-то чего-то не хватает и т.д. Если вы для графического представления требований будете использовать общепринятую нотацию (например, уже упомянутый UML), вы получите дополнительные преимущества: вашу схему смогут без труда понимать и дорабатывать коллеги, а в итоге может получиться хорошее дополнение к текстовой форме представления требований.

Рекомендованные практические статьи:

Тестировщики и требования: непараллельные прямые

<http://software-testing.ru/library/around-testing/requirements/1399-testers-and-requirements>

Тестирование без требований

http://qaclub.com.ua/wp-content/uploads/d/qaclub11_23-07-10/Shapoval-Testing_without_requirements.pdf

Типичные ошибки при анализе и тестировании требований (Куликов)

http://okiseleva.blogspot.com/2015/10/blog-post_8.html

1.6. Архитектура компьютера.

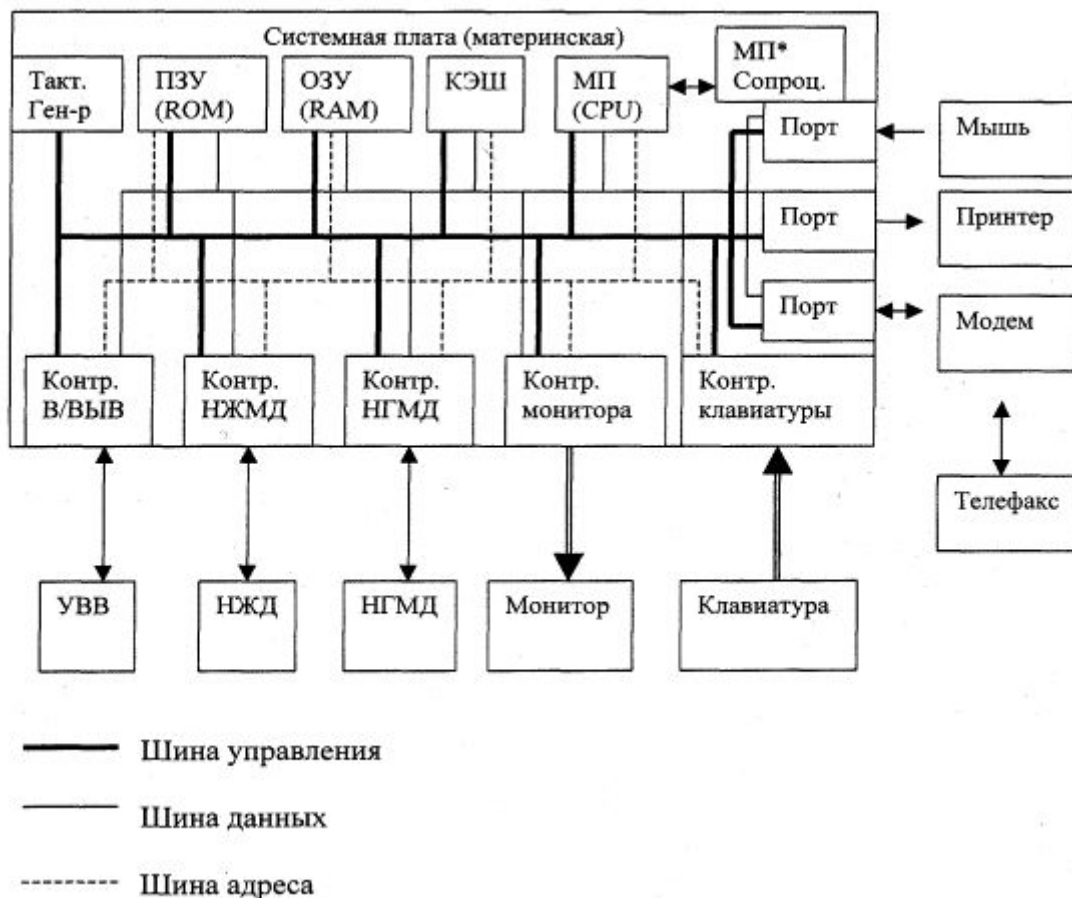
При изучении таких видов тестирования, как: нагрузочное, быстродействия, стабильности и т.д. мы сталкиваемся с такими понятиями, как процессорное время, оперативная память, жесткий диск. Инженер обязан хорошо понимать, из чего состоит компьютер и какие функции выполняет тот или иной узел.

Основной принцип построения современных компьютеров носит название **архитектуры фон Неймана** - в честь американского ученого венгерского происхождения Джона фон Неймана, который ее предложил.

Современную архитектуру компьютера определяют следующие принципы:

1. **Принцип программного управления.** Обеспечивает автоматизацию процесса вычислений на ЭВМ. Согласно этому принципу, для решения каждой задачи составляется программа, которая определяет последовательность действий компьютера. Эффективность программного управления будет выше при решении задачи этой же программой много раз (хотя и с разными начальными данными).
2. **Принцип программы, сохраняемой в памяти.** Согласно этому принципу, команды программы подаются, как и данные, в виде чисел и обрабатываются так же, как и числа, а сама программа перед выполнением загружается в оперативную память, что ускоряет процесс ее выполнения.
3. **Принцип произвольного доступа к памяти.** В соответствии с этим принципом, элементы программ и данных могут записываться в произвольное место оперативной памяти, что позволяет обратиться по любому заданному адресу (к конкретному участку памяти) без просмотра предыдущих.

На основании этих принципов можно утверждать, что современный компьютер - техническое устройство, которое после ввода в память начальных данных в виде цифровых кодов и программы их обработки, также выраженной цифровыми кодами, способно автоматически осуществить вычислительный процесс, заданный программой, и выдать готовые результаты решения задачи в форме, пригодной для восприятия человеком.



Основные узлы компьютера следующие:

Основная часть системной платы — **микропроцессор (МП)** или **CPU (Central Processing Unit)**, он управляет работой всех узлов ПК и программой, описывающей алгоритм решаемой задачи. МП имеет сложную структуру в виде электронных логических схем. В качестве его компонентов можно выделить:

- 1) **АЛУ** — арифметико-логическое устройство, предназначенное для выполнения арифметических и логических операций над данными и адресами памяти;
- 2) **Регистры** или **микропроцессорная память** — сверхоперативная память, работающая со скоростью процессора, АЛУ работает именно с ними;
- 3) **УУ** - устройство управления — управление работой всех узлов МП посредством выработки и передачи другим его компонентам управляющих импульсов, поступающих от кварцевого тактового генератора, который при включении ПК

начинает вибрировать с постоянной частотой (100 МГц, 200-400 МГц). Эти колебания и задают темп работы всей системной платы;

- 4) **СПр** - система прерываний — специальный регистр, описывающий состояние МП, позволяющий прерывать работу МП в любой момент времени для немедленной обработки некоторого поступившего запроса, или постановки его в очередь. После обработки запроса СПр обеспечивает восстановление прерванного процесса;

- 5) **Устройство управления общей шиной** — интерфейсная система.

Для расширения возможностей ПК и повышения функциональных характеристик микропроцессора дополнительно может поставляться математический сопроцессор, служащий для расширения набора команд МП. Например, математический сопроцессор IBM-совместимых ПК расширяет возможности МП для вычислений с плавающей точкой; сопроцессор в локальных сетях (LAN-процессор) расширяет функции МП в локальных сетях.

Характеристики процессора:

- **быстродействие** (производительность, тактовая частота) — количество операций, выполняемых в секунду.
- **разрядность** — максимальное количество разрядов двоичного числа, над которыми одновременно может выполняться машинная операция.

Пример: Первый процессор был 4-разрядным, то есть работал с числами, представляемыми 4 двоичными разрядами - $2^4 = 16$ чисел, 16 адресов.

16-разрядный процессор одновременно может работать с $2^{16} = 65536$ числами и адресами. 32-разрядный - $2^{32} = 4\,294\,967\,296$ чисел.

При тактовой частоте 33 МГц обеспечивается выполнение 7 млн. коротких машинных операций (сложение, вычитание, пересылка информации); при частоте 100 МГц - 20 млн. аналогичных операций.

Интерфейсная система - это:

- **шина управления (ШУ)** - предназначена для передачи управляющих импульсов и синхронизации сигналов ко всем устройствам ПК;

- **шина адреса** (ША) - предназначена для передачи кода адреса ячейки памяти или порта ввода/вывода внешнего устройства;
- **шина данных** (ШД) - предназначена для параллельной передачи всех разрядов числового кода;
- **шина питания** - для подключения всех блоков ПК к системе электропитания.

Интерфейсная система обеспечивает три направления передачи информации:

- между МП и оперативной памятью;
- между МП и портами ввода/вывода внешних устройств;
- между оперативной памятью и портами ввода/вывода внешних устройств.

Обмен информацией между устройствами и системной шиной происходит с помощью кодов ASCII.

Память - устройство для хранения информации в виде данных и программ. Память делится прежде всего на внутреннюю (расположенную на системной плате) и внешнюю (размещенную на разнообразных внешних носителях информации).

Внутренняя память в свою очередь подразделяется на:

1. ПЗУ (постоянное запоминающее устройство) или **ROM** (read only memory), которое содержит постоянную информацию, сохраняемую даже при отключенном питании и которая служит для тестирования памяти и оборудования компьютера, начальной загрузки ПК при включении. Запись на специальную кассету ПЗУ происходит на заводе фирмы-изготовителя ПК и несет черты его индивидуальности. **Объем** ПЗУ относительно невелик - от 64 до 256 Кб.

2. ОЗУ (оперативное запоминающее устройство, **ОП** — оперативная память) или **RAM** (random access memory), служит для оперативного хранения программ и данных, сохраняемых только на период работы ПК. Она энергозависима, при отключении питания информация теряется. ОП выделяется особыми функциями и спецификой доступа:

- ОП хранит не только данные, но и выполняемую программу;
- МП имеет возможность прямого доступа в ОП, минуя систему ввода/вывода.

Логическая организация памяти — адресация, размещение данных определяется ПО, установленным на ПК, а именно ОС.

3. Кэш-память - имеет малое время доступа, служит для временного хранения промежуточных результатов и содержимого наиболее часто используемых ячеек ОП и регистров МП.

4. Внешняя память. Устройства внешней памяти весьма разнообразны.

Предлагаемая классификация учитывает тип **носителя**, т.е. материального объекта, способного хранить информацию. Это жесткие диски, SSD.

Данные на дисках хранятся в **файлах** — именованных областях внешней памяти, выделенных для хранения массива данных. Кластеры, выделяемые файлу, могут находиться в любом свободном месте дисковой памяти и не обязательно являются смежными. Вся информация о том, где именно записаны кусочки файла, хранится в **таблице размещения файлов FAT** (file allocation table).

5. Контроллеры служат для обеспечения прямой связи с ОП, минуя МП, они используются для устройств быстрого обмена данными с ОП - накопители (жесткий диск), дисплей и др., обеспечения работы в групповом или сетевом режиме. Клавиатура, дисплей, мышь являются медленными устройствами, поэтому они связаны с системной платой контроллерами и имеют в ОП свои отведенные участки памяти.

6. Порты бывают входными и выходными, универсальными (ввод - вывод), они служат для обеспечения обмена информацией ПК с внешними, не очень быстрыми, устройствами. Информация, поступающая через порт, направляется в МП, а потом в ОП. Выделяют два вида портов:

- **последовательный** — обеспечивает побитовый обмен информацией, обычно к такому порту подключают модем;
- **параллельный** — обеспечивает побайтный обмен информацией, к такому порту подключают принтер. Современные ПК обычно оборудованы 1 параллельным и 2 последовательными портами.

Конкретный набор компонентов, входящих в данный компьютер, называется его **конфигурацией**. **Минимальная конфигурация ПК**, необходимая для его работы, включает в себя системный блок (там находятся МП, ОП, ПЗУ, НЖМД, НГМД),

клавиатуру (как устройство ввода информации) и монитор (как устройство вывода информации).

УРОК 2.

2.1. Уровни тестирования.

Уровень тестирования определяет то, **над чем производятся тесты**: над отдельным модулем, группой модулей или системой в целом.

*Уровни **отличаются** в первую очередь ролями и навыками пользователей, которые лучше всего подходят для тестирования.* На разных уровнях тестированию подлежат разные объекты. Возможна путаница, вызванная тем, что единого общепринятого набора классификаций не существует. То есть “По уровню детализации приложения” это тоже самое что и “по уровню тестирования”. Тестирование на разных уровнях производится на протяжении всего жизненного цикла разработки и сопровождения программного обеспечения. Очень важно обеспечить хороший баланс между различными уровнями тестирования. Проведение тестирования на всех уровнях системы - это залог успешной реализации и сдачи проекта.

В Тестировании ПО можно выделить 4 типичных уровня Тестирования:

- Модульное тестирование
- Интеграционное тестирование
- Системное тестирование
- Приемочное тестирование

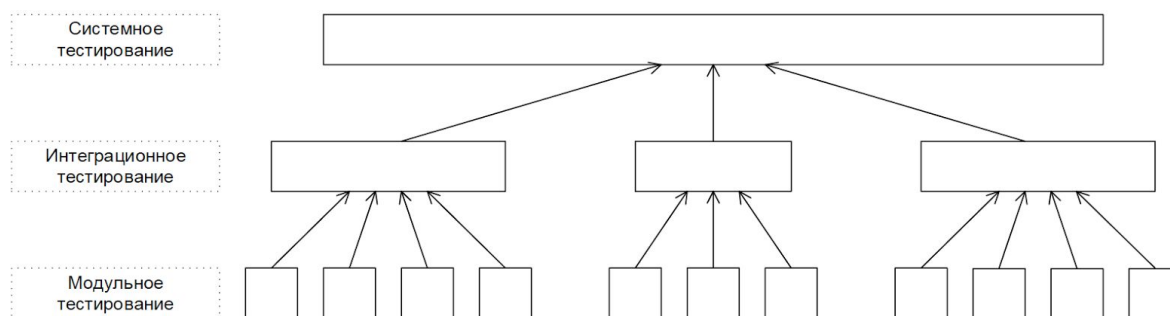
Модульное (компонентное) тестирование (Module testing, Component testing) направлено на проверку отдельных небольших частей приложения, которые можно исследовать изолированно от других подобных частей. Часто данный вид тестирования реализуется с использованием специальных технологий и инструментальных средств автоматизации тестирования, значительно упрощающих и ускоряющих разработку соответствующих тест-кейсов.

При выполнении данного тестирования могут проверяться отдельные функции или методы классов, сами классы, взаимодействие классов, небольшие библиотеки, отдельные части приложения.

Интеграционное тестирование (Integration Testing) направлено на проверку взаимодействия между несколькими частями приложения, каждая из которых, в свою очередь, проверена отдельно на стадии модульного тестирования. К сожалению, даже если мы работаем с очень качественными отдельными компонентами, «на стыке» их взаимодействия часто возникают проблемы. Именно эти проблемы и выявляет интеграционное тестирование.

Системное тестирование (System Testing) направлено на проверку всего приложения как единого целого, собранного из частей, проверенных на двух предыдущих стадиях. Здесь не только выявляются дефекты «на стыках» компонентов, но и появляется возможность полноценно взаимодействовать с приложением с точки зрения конечного пользователя, применяя множество других видов тестирования.

Для лучшего запоминания степень детализации в модульном, интеграционном и системном тестировании показана схематично на рисунке:



Приемочное тестирование (Acceptance Testing) это последняя разновидность тестирования, выполняемого до развертывания программного обеспечения. Цель приемки - определение, удовлетворяет ли система приемочным критериям, и вынесения решения заказчиком или другим уполномоченным лицом - принимается приложение или нет. Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

Последовательность и время выполнения разных уровней тестирования

Принято считать, что тестирование компонентов нужно выполнять в первую очередь, и приступать к другим разновидностям тестов только после успешного завершения этого вида тестирования. Однако такой подход, вообще говоря, неверен для итерационного процесса разработки. Вместо этого рекомендуется выявлять тесты компонентов, интеграции и системы, которые с максимальной вероятностью позволят обнаружить ошибки, и выполнять эти тесты по принципу максимального риска.

2.2. Юнит тесты

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Модуль, в контексте модульного тестирования, представляет собой минимальную смысловую единицу исходного кода, пригодную для тестирования (функция, процедура, метод, в отдельных случаях – класс). Модульный тест проверяет работоспособность модуля в условиях изоляции от других модулей приложения. Исключив влияние других модулей, мы проверяем наш модуль в стерильных условиях: если результат выполнения модульного теста не соответствует ожидаемому, то ошибка кроется в логике нашего модуля, а не во внешних факторах.

Модульные тесты и TDD

Изначально, основным назначением модульного тестирования были регрессионные проверки, но, как оказалось, unit-тесты способны на большее.

Вторым рождением модульные тесты обязаны череде нововведений в методиках создания ПО. При переходе к гибким методологиям разработки возник вопрос - как сделать каждую итерацию максимально эффективной? Не делать ничего лишнего. Есть требование к продукту, его надо реализовать с минимальными затратами времени и сил. Для этого разработчику нужно четко представлять конечный результат

на языке программного кода. Один из вариантов такого представления – модульные тесты, сердце TDD (Test driven development - разработка через тестирование) . Разработчик пишет тесты на новую функциональность. Первоначально прогон каждого теста завершается неудачно (тест существует, но кода, который он проверяет, нет). Разработчик пишет код приложения и снова прогоняет тест. Если тест пройден успешно (равно как и все ранее написанные тесты на “старую” функциональность), промежуточная цель считается достигнутой.

Достоинства модульных тестов:

1. Лучшее качество кода. По сути, комбинация “основного” кода и модульных тестов – это двойная запись одной и той же функциональности. В коде могут быть ошибки. В тесте могут быть ошибки. Но вероятность ошибки и в коде и в тесте гораздо меньше.
2. В случае TDD: ожидаемый результат выполнения кода. Модульные тесты описывают прототип приложения – заложенную структуру, базовые алгоритмы. Соответственно, код реализует идеи разработчика, выраженные в модульных тестах (перевод бизнес-языка в язык программного кода).
3. В случае TDD: возможность не делать лишних движений. Модульные тесты соответствуют бизнес-требованиям → код прошел модульные тесты → задача выполнена.
4. Возможность проводить рефакторинг без опасений “поломать” работу приложения. Основное назначение модульных тестов состоит не столько в том, чтобы находить баги в текущей версии продукта, сколько в том, чтобы не пропустить баги в старой функциональности при добавлении новой. К примеру, маленькое “улучшение” в алгоритме шифрования не разрушит всю систему, если на алгоритм и смежные компоненты есть модульные тесты.
5. Скорость нахождения багов. По оценкам экспертов, модульные тесты позволяют найти около 15% багов, выявленных в ходе полного цикла разработки. В то же время, большей частью это критические баги, и их раннее выявление способно сберечь существенную долю ресурсов и предотвратить катаклизмы в среде пользователя программного продукта. Чем короче путь в исправлении бага, тем меньше будет затрачено усилий на его исправление.
6. Возможность тестирования базовой функциональности без UI и отладки без использования “живой” системы.

7. Более удобная координация работы распределенной группы программистов. Модульные тесты позволяют провести приемочные испытания перед интегрированием модулей в единую систему. Модульные тесты – это мера ответственности разработчика. Если разработчик выпускает код, который не проходит модульные тесты, это может замедлить продвижение всего проекта. Представим, что в системе обслуживания телескопов есть код, который должен выполняться только в день летнего равноденствия. Если программист, написавший этот код, не подготовит для него тест, то следующий программист может внести правку и не проверить работоспособность кода в этот единственный день в году.
8. Модульные тесты служат дополнением к документации и помогают новому разработчику войти в курс дела. Объектно-ориентированные языки программирования предполагают наличие зависимостей между классами. Небольшое изменение в, казалось бы, второстепенном классе, способно разрушить работу всей системы, поэтому модульные тесты в таких случаях могут оказаться очень полезны.
9. Точечная настройка производительности системы и обнаружение утечек памяти. Воспроизвести 100 тыс. одновременных подключений и выполнение определенной последовательности действий в системе не так-то просто. Гораздо удобнее воспроизвести эту нагрузку с помощью модульных тестов и наблюдать поведение продукта с помощью `purify`, `valgrind` и подобных инструментов. Можно пойти дальше и рассматривать результаты замера производительности как критерий оценки внесенных изменений. Если при добавлении новой функциональности производительность системы ухудшилась, то это может служить “красным светом” для внедрения данной версии продукта.

Недостатки модульных тестов

1. Временные затраты. Вместо написания кода, который можно продать, разработчик будет трудиться над тестами. Сама идея предохраняться от всего и вся может оказаться надуманной. Лучше по факту посмотреть, что сломалось, и чинить именно эту часть.
2. Модульных тестов недостаточно для качественного тестирования приложений. Ясно, что модульные тесты не охватывают работу всей системы в целом.
3. Модульные тесты выполняются “в вакууме”, в стерильных условиях. В реальной жизни код может валиться, скажем, из-за проблем с

производительностью, причем именно в том месте, где, казалось бы, модульные тесты и не нужны.

4. Модульные тесты, как и любые другие, покажут наличие ошибок, но не докажут их отсутствие.

Рассмотрим пример теста на языке Python:

Допустим, у нас есть функция `multiply`, которая выдает нам произведение двух чисел.

Юнит тесты будут выглядеть следующим образом:

```
import unittest

from unnecessary_math import multiply


class TestUM(unittest.TestCase):

    def setUp(self):

        pass

    def test_numbers_3_4(self):

        self.assertEqual( multiply(3,4), 12)

    def test_numbers_0_3(self):

        self.assertEqual( multiply(0,3), 0)

    def test_numbers_-1_3(self):

        self.assertEqual( multiply(-1,3), -3)


if __name__ == '__main__':

    unittest.main()
```


При запуске тестов будет произведена проверка трех случаев - произведение двух положительных чисел, нуля и положительного, отрицательного и положительного.

Допустим, при рефакторинге разработчик перепутает требования, и решит, что на выход функции не должен подаваться 0, и нужно выдавать в таком случае ошибку пользователю (допустим, перепутает с функцией расчета скидки, для которой при введении 0 нужно выдать пользователю сообщение, что скидка введена неверно). После таких изменений тест `test_numbers_0_3` выдаст ошибку, так как результатом будет не 0, а сообщение об ошибке. Это позволит разработчику на ранней стадии отловить проблему.

Подробнее о юнит тестах:

<http://blog.openquality.ru/unit-tests-why>

<http://rstdn.org/article/testing/UnitTesting.xml>

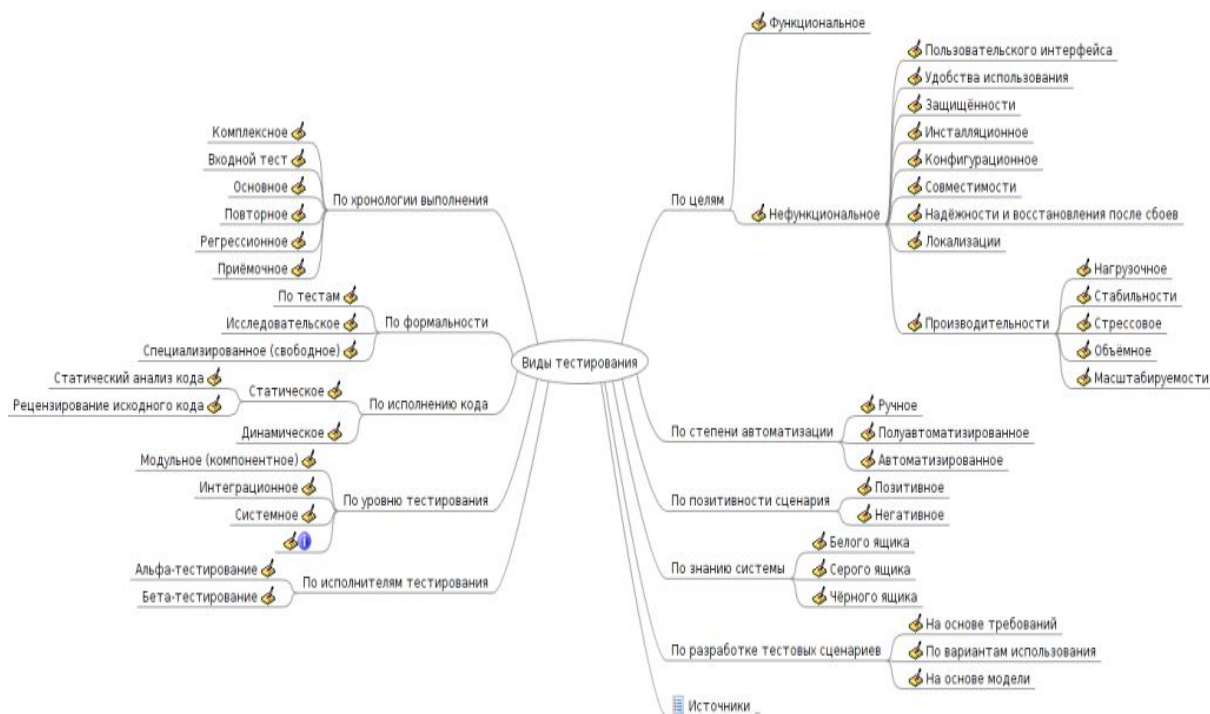
<http://gahcep.github.io/blog/2013/02/10/qa-in-python-unittest/>

<http://geosoft.no/development/unittesting.html>

2.3. Классификация и Виды тестирования + Ad-hoc, Error Guessing, Exploratory testing + Перцепсионное тестирование

Тестирование можно классифицировать по очень большому количеству признаков. Существует несколько признаков, по которым принято производить классификацию типов и видов тестирования.

Тестирование существенно различается по задачам, которые с их помощью решаются, и по используемой технике. Различие задач тестирования приводит, естественным образом, к необходимости использовать весьма разнообразные типы (виды) тестирования.



Из схемы выше легко убедиться, что материал достаточно объёмен и сложен, а глубокое понимание каждого пункта в классификации требует определенного опыта, поэтому мы рассмотрим самый простой, минимальный набор информации, необходимый начинающему тестировщику.

Используйте нижеприведенный список как очень краткую «шпаргалку для запоминания». Итак, тестирование можно классифицировать:

1. По запуску кода на исполнение:

- **Статическое тестирование (Static Testing)** — без запуска кода. В рамках этого подхода тестированию могут подвергаться документы, графические прототипы, код приложения и т.д.
- **Динамическое тестирование (Dynamic Testing)** — тестирование с запуском кода на исполнение. Основная идея состоит в том, что проверяется реальное поведение (части) приложения.

2. По доступу к коду и архитектуре приложения:

- **Метод черного ящика (Black Box Testing)** — доступа к коду нет.

Тестирование как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы. Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Целью этой техники является поиск ошибок в неправильно реализованные или недостающие функции, ошибки интерфейса, ошибки в структурах данных или организации доступа к внешним базам данных, ошибки поведения или недостаточная производительности системы.

Пример: Тестировщик проводит тестирование веб-сайта, не зная особенностей его реализации, используя только предусмотренные разработчиком поля ввода и кнопки. Источник ожидаемого результата – спецификация.

- **Метод белого ящика (White Box Testing)** — доступ к коду есть.

Данный метод предполагает, что внутренняя структура/устройство/реализация системы известны тестировщику. Мы выбираем входные значения, основываясь на знании кода, который будет их обрабатывать. Точно так же мы знаем, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации – обязательны для этой техники. Тестирование белого ящика – углубление во внутреннее устройство системы, за пределы ее внешних интерфейсов.

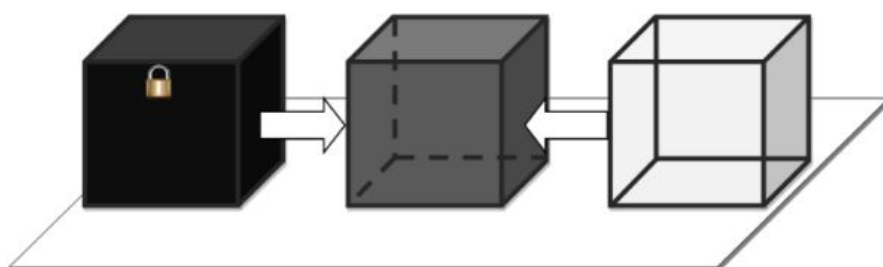
Пример: Тестировщик, который, как правило, является программистом, изучает реализацию кода поля ввода на веб-странице, определяет все предусмотренные (как правильные, так и неправильные) и не предусмотренные пользовательские вводы, и сравнивает фактический результат выполнения программы с ожидаемым. При этом ожидаемый результат определяется именно тем, как должен работать код программы.

- **Метод серого ящика (Grey Box Testing)** — к части кода доступ есть, к части — нет.

Метод тестирования программного обеспечения, который предполагает, комбинацию White Box и Black Box подходов. То есть, внутреннее устройство программы нам известно лишь частично. Предполагается, например, доступ к внутренней структуре и алгоритмам работы ПО для написания максимально эффективных тест-кейсов, но само тестирование проводится с помощью техники черного ящика, то есть, с позиции

пользователя. Эту технику тестирования также называют методом полупрозрачного ящика: что-то мы видим, а что-то – нет.

Пример: Тестировщик изучает код программы с тем, чтобы лучше понимать принципы ее работы и изучить возможные пути ее выполнения. Такое знание поможет написать тест-кейс, который наверняка будет проверять определенную функциональность.



3. По степени автоматизации:

- **Ручное тестирование (Manual Testing)** — тест-кейсы выполняет человек.

Тестировщики вручную выполняют тесты, не используя никаких средств автоматизации. Ручное тестирование – самый низкоуровневый и простой тип тестирования, не требующих большого количества дополнительных знаний.

- **Автоматизированное тестирование (Automated Testing)** — тест-кейсы частично или полностью выполняет специальное инструментальное средство.

Предполагает использование специального программного обеспечения (помимо тестируемого) для контроля выполнения тестов и сравнения ожидаемого фактического результата работы программы. Этот тип тестирования помогает автоматизировать часто повторяющиеся, но необходимые для максимизации тестового покрытия задачи.

- **Полуавтоматизированное (Partially automated)**

Один из фундаментальных принципов тестирования гласит: 100% автоматизация невозможна. Поэтому, ручное тестирование – необходимость.

4. По уровню детализации приложения (по уровню тестирования):

- **Модульное (компонентное) тестирование (Module testing, Component testing)** — проверяются отдельные небольшие части приложения.
- **Интеграционное тестирование (Integration Testing)** — проверяется взаимодействие между несколькими частями приложения.
- **Системное тестирование (System Testing)** — приложение проверяется как единое целое.

5. По принципам работы с приложением:

- **Позитивное тестирование** — все действия с приложением выполняются строго по инструкции без никаких недопустимых действий, некорректных данных и т.д.
- **Негативное тестирование** — в работе с приложением выполняются (некорректные) операции и используются данные, потенциально приводящие к ошибкам (классика жанра — деление на ноль).

Негативные тесты НЕ предполагают возникновения в приложении ошибки. Напротив — они предполагают, что верно работающее приложение даже в критической ситуации поведёт себя правильным образом (в примере с делением на ноль, например, отобразит сообщение «Делить на ноль запрещено»).

6. По степени подготовленности

- **Тестирование по документации**
- **Интуитивное тестирование (Ad-hoc)**

7. По объекту тестирования:

- **Функциональное (functional testing)**

Функциональные тесты базируются на функциях и особенностях, а также взаимодействии с другими системами, и могут быть представлены на всех

уровнях тестирования: *компонентном или модульном (Component/Unit testing)*, *интеграционном (Integration testing)*, *системном (System testing)* и *приемочном (Acceptance testing)*. Функциональные виды тестирования рассматривают внешнее поведение системы. Далее перечислены одни из самых распространенных видов функциональных тестов:

- **Функциональное тестирование (Functional testing)** рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.
 - **Тестирование безопасности (Security and Access Control Testing)** — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.
 - **Тестирование взаимодействия (Interoperability Testing)** — это функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование (integration testing).
-
- **Нефункциональное (non-functional testing)** описывает тесты, необходимые для определения характеристик программного обеспечения, которые могут быть измерены различными величинами. В целом, это тестирование того, "Как" система работает. Далее перечислены основные виды нефункциональных тестов:
 - Все виды **тестирования производительности**:
 - *нагрузочное тестирование (Performance and Load Testing)* — это автоматизированное тестирование, имитирующее работу определенного количества бизнес пользователей на каком-либо общем (разделяемом ими) ресурсе.
 - *стрессовое тестирование (Stress Testing)* — позволяет проверить насколько приложение и система в целом работоспособны в условиях стресса и также оценить способность системы к регенерации, т.е. к возвращению к нормальному состоянию после прекращения воздействия стресса.

- *тестирование стабильности или надежности (Stability / Reliability Testing)* — проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
- *объемное тестирование (Volume Testing)* — получение оценки производительности при увеличении объемов данных в базе данных приложения.
- **Тестирование установки (Installation testing)** — направлено на проверку успешной инсталляции и настройки, а также обновления или удаления программного обеспечения.
- **Тестирование удобства пользования (Usability Testing)** — это метод тестирования, направленный на установление степени удобства использования, обучаемости, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий. [ISO 9126]
- **Тестирование на отказ и восстановление (Failover and Recovery Testing)** — проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети).
- **Конфигурационное тестирование (Configuration Testing)** — специальный вид тестирования, направленный на проверку работы программного обеспечения при различных конфигурациях системы (заявленных платформах, поддерживаемых драйверах, при различных конфигурациях компьютеров и т.д.)
- **Связанные с изменениями виды тестирования:**
 - **Дымовое тестирование (Smoke Testing, Build Verification Testing)**

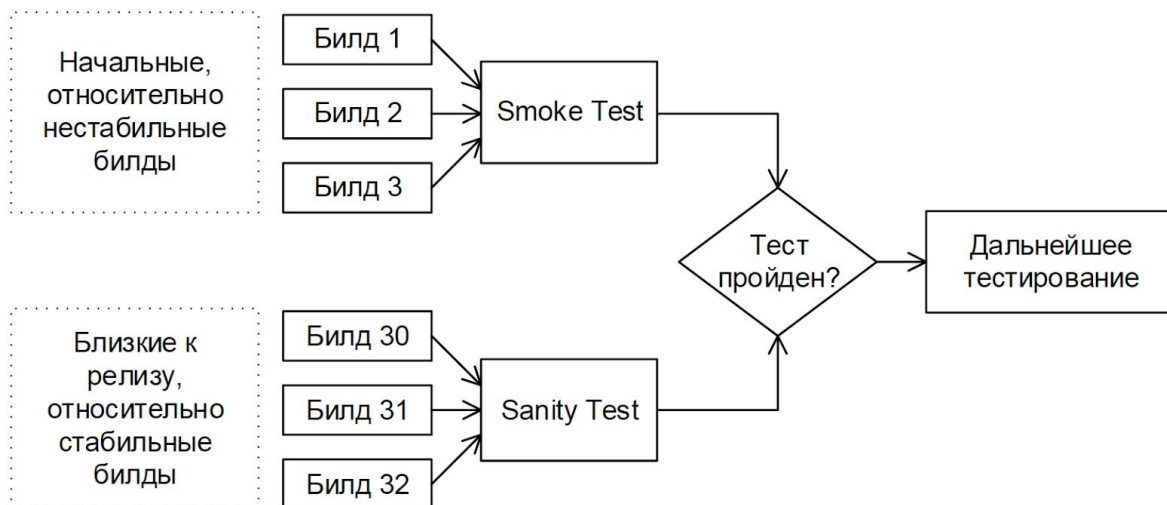
Проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной саму идею использования приложения. Первый прогон программы (после написания или после внесения существенных изменений). Как правило, используется для

определения, готова ли программа для проведения более обширного тестирования.

- **Санитарное тестирование или проверка согласованности/исправности (Sanity Testing)**

Это узконаправленное тестирование достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям

Важно: в некоторых источниках ошибочно полагают, что санитарное и дымовое тестирование - это одно и то же. Мы же полагаем, что эти виды тестирования имеют "вектора движения", направления в разные стороны. В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено **вглубь** проверяемой функции, в то время как дымовое направлено **вширь**, для покрытия тестами как можно большего функционала в кратчайшие сроки.



- **Регрессионное тестирование (Regression Testing)**

Тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования. Регрессионное тестирование является неотъемлемым инструментом обеспечения качества и активно используется практически в любом проекте.

- **Повторное тестирование (Re-testing, Confirmation testing)**

Выполнение тест-кейсов, которые ранее обнаружили дефекты, с целью подтверждения устранения дефектов. Фактически этот вид тестирования сводится к действиям на финальной стадии жизненного цикла отчета о дефекте, направленным на то, чтобы перевести дефект в состояние «проверен» и «закрит».

УРОК 3.

3.1. Системы контроля версий.

Сначала простой пример - инженер работает сам над документом и хочет сохранять версии, чтобы ничего не потерять. Один из вариантов решения - копировать вручную папку и добавлять к названию дату или номер версии (думаю многие делали так с дипломными работами, курсовыми и другими важными документами). Но существуют гораздо более удобные инструменты.

При работе над созданием ПО (либо любой другой работе с документами) возникает проблема: как нескольким инженерам работать одновременно над одним и тем же продуктом? Как не потерять информацию, не поломать ранее работающий функционал своими изменениями?

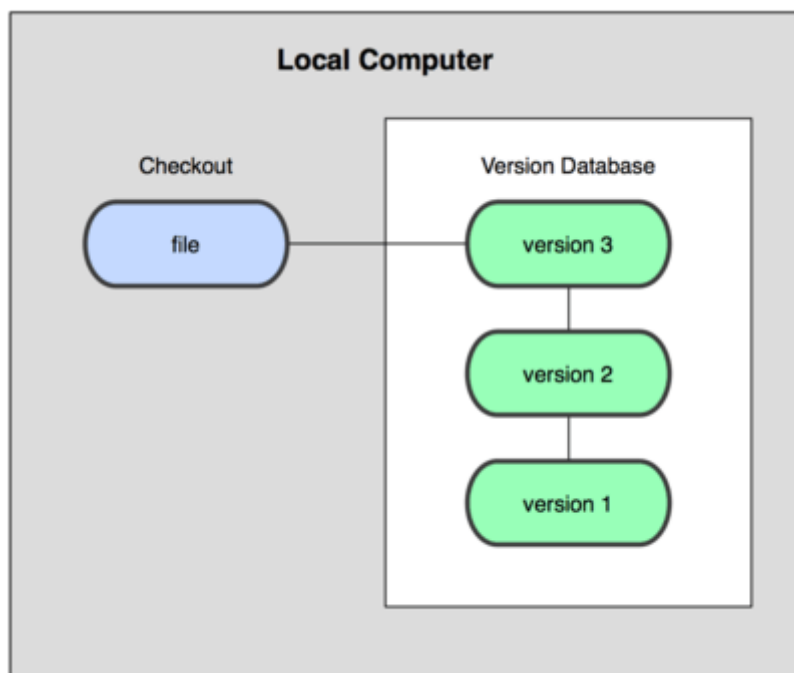
Проект программного продукта может содержать сотни файлов кода, и при работе над одной задачей разработчик может менять десятки из них. Как разработчикам синхронизироваться между собой и иметь возможность одновременно работать с одними и теми же файлами?

Для решения этих задач существуют Системы контроля версий.

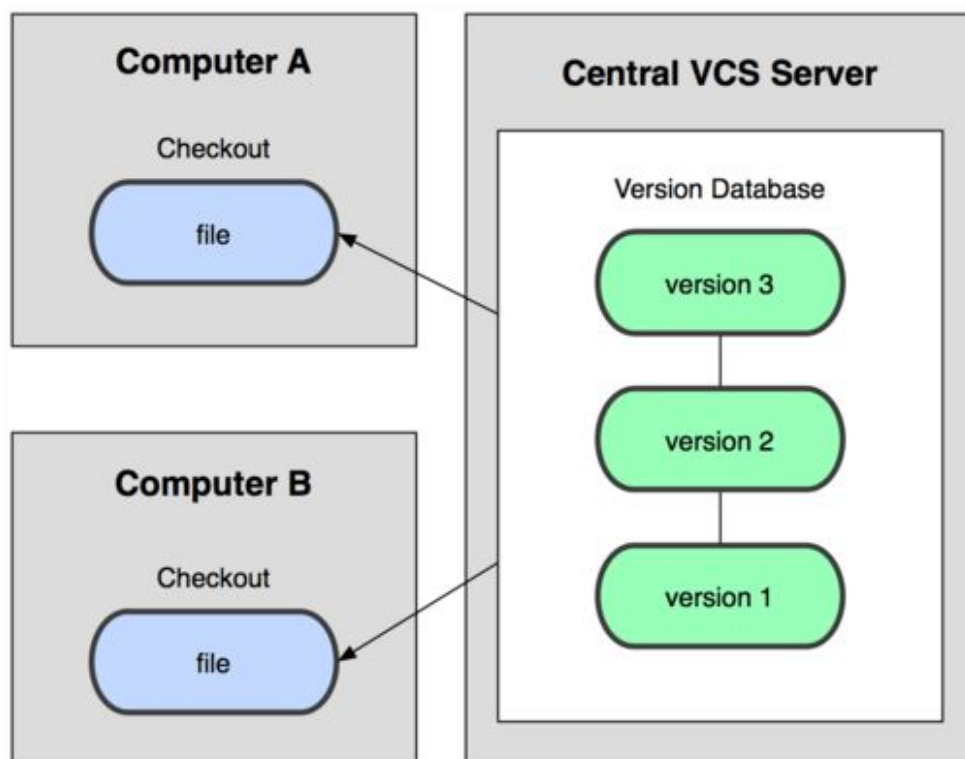
Система контроля (управления) версий (*Version Control System, VCS* или *Revision Control System*) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения **исходных кодов** разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

Для решения первой проблемы (версионности файла на одной машине) существуют локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов:



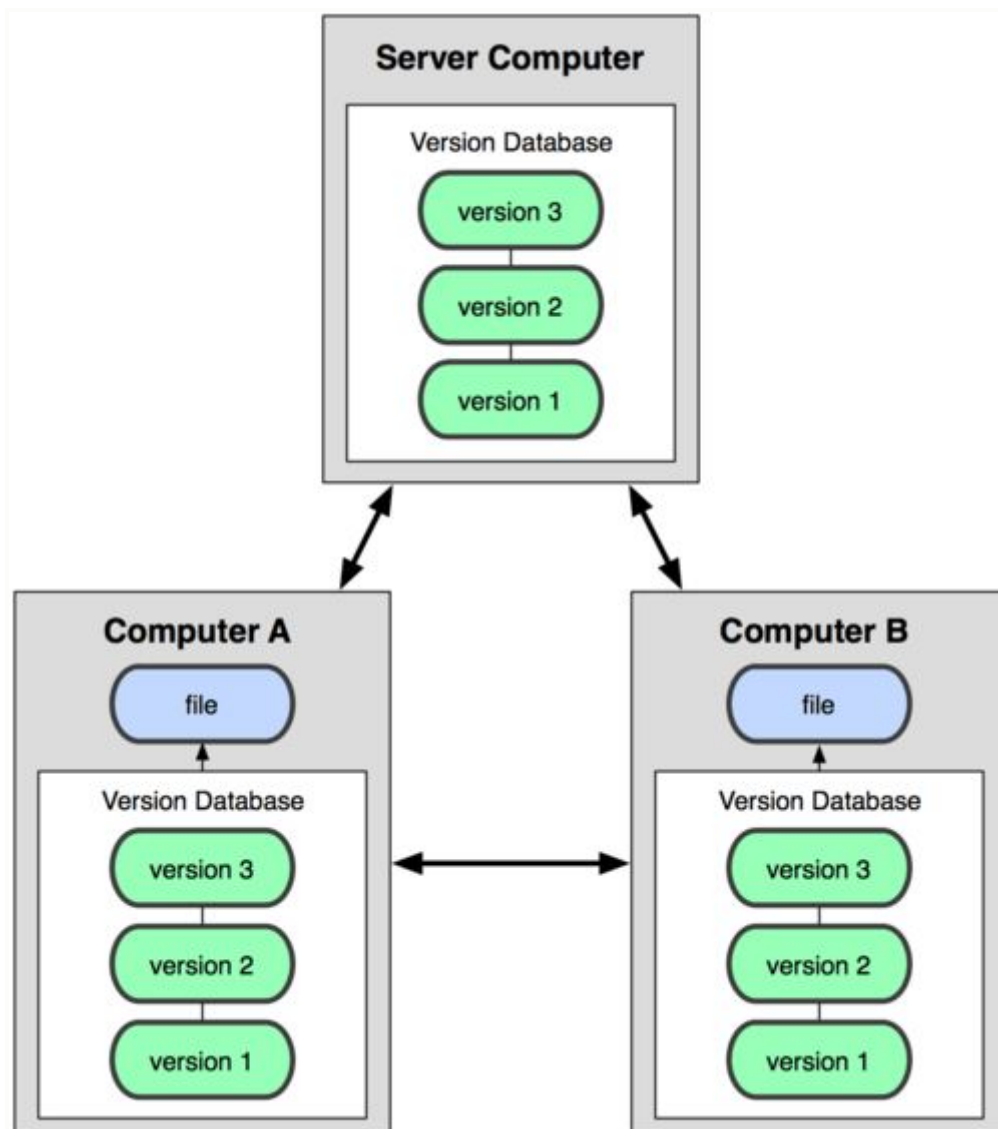
Для решения же второй проблемы (работы множества разработчиков над одним продуктом) были созданы централизованные системы контроля версий (ЦСКВ). В таких системах, например CVS, Subversion и Perforce, есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него. Много лет это было стандартом для систем контроля версий (см. рис. 1-2)



Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы. Если же повреждается диск с центральной базой данных и нет резервной копии, вы теряете абсолютно всё — всю историю проекта, разве что за исключением нескольких рабочих версий, сохранившихся на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

И в этой ситуации в игру вступают распределённые системы контроля версий (РСКВ). В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создает себе полную копию всех данных.



Для начала, рассмотрим подробнее централизованные системы контроля версий (ЦСКВ). Основной принцип работы таких систем - основная версия документов хранится на удаленном сервере (репозиторий). Каждый инженер может загрузить себе с сервера документы. Таким образом у него будет локальная копия, с которой он сможет работать. По окончании работы, инженер должен сохранить свои изменения в основную версию - на сервер. Таким образом другие смогут скачать уже новую версию, содержащую изменения инженера, и работать с ней.

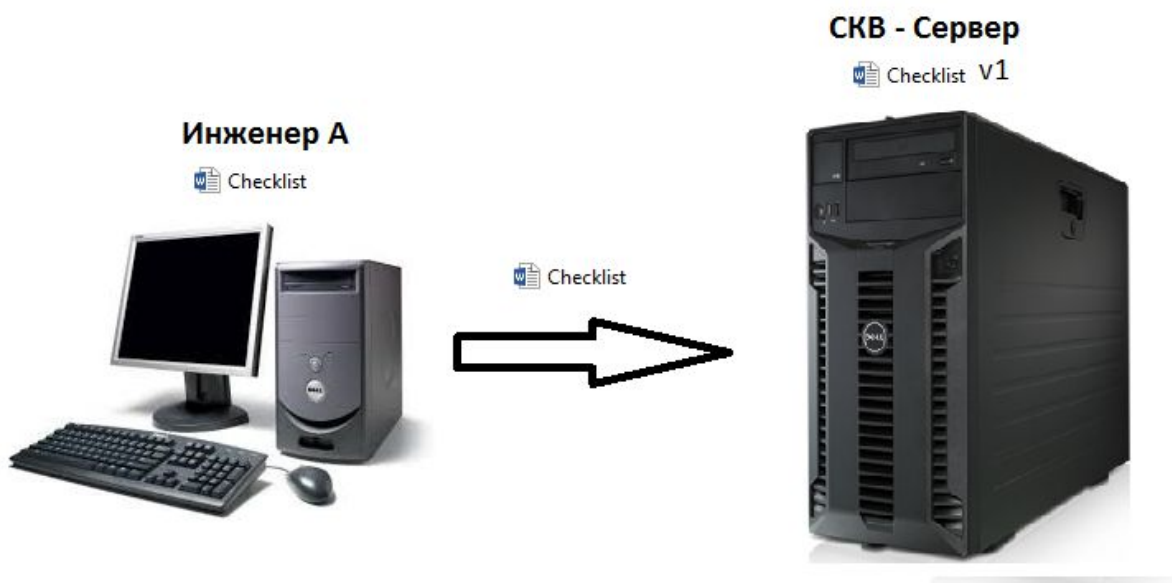
Рассмотрим на примере одного файла и системы контроля версий **Subversion (SVN)** (в реальной жизни речь идет о директориях и множестве файлов). Тестировщик А создал файл, содержащий чеклист на своем локальном компьютере:

Checklist.doc

Чтобы другие члены команды получили доступ к этому файлу, он сохраняет его в репозиторий - файл копируется на сервер и получает номер версии v1.

Это делается с помощью команд:

- **svn add <файл/папка>** — включить файл/папку в локальную копию проекта
- **svn commit** — отправляет все изменения локальной копии в репозиторий.



Теперь любой человек, имеющий доступ к репозиторию, может его скачать и работать с ним (если до сих пор репозиторий не был скачан):

- **svn co http://.... (checkout)** — скачивает репозиторий с сервера на локальную машину.



Теперь, когда инженер В внесет свои изменения и сохранит файл в репозиторий - файл получит следующую версию - v2.

Если же теперь инженер А захочет обновить файл (получить v2), он должен обновить свой локальный репозиторий:

- **svn update** — обновляет содержимое локальной копии до самой последней версии из репозитория.

Если же инженерам А и В необходимо одновременно работать с файлом Checklist.doc, в VCS есть две модели, которые позволяют избегать этой проблемы:

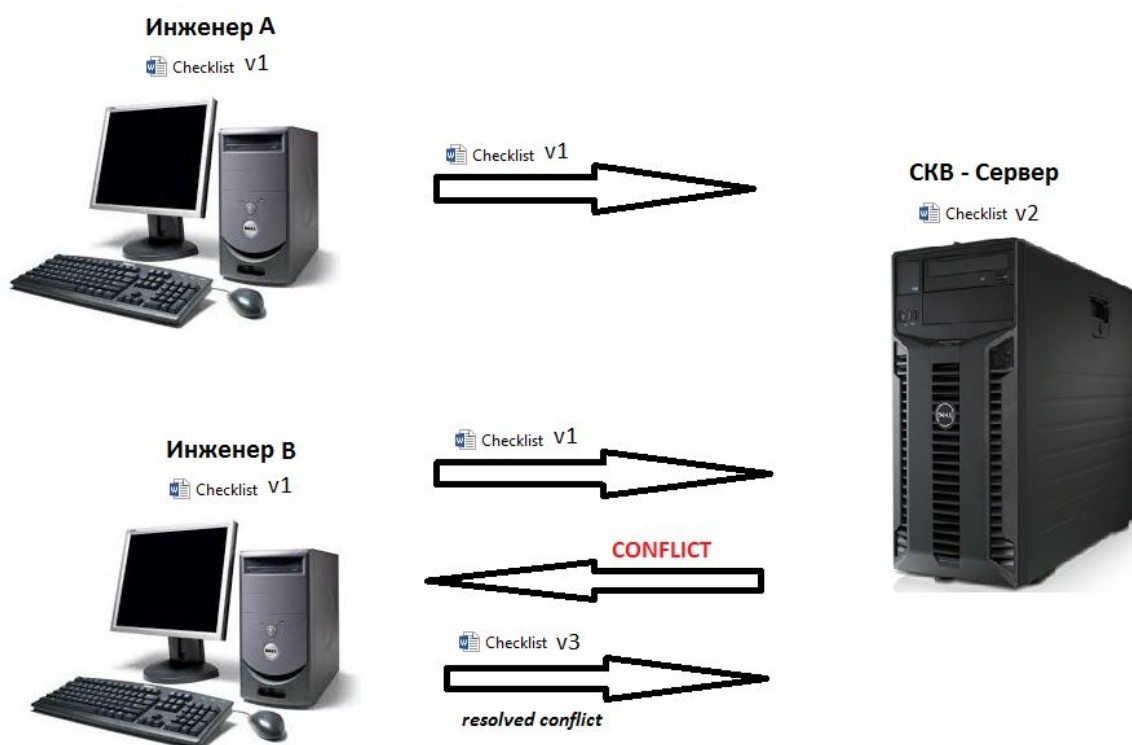
1. Блокировка — изменение — разблокировка.

- Согласно этой модели, когда кто-либо начинает работу с файлом, этот файл блокируется, и все остальные пользователи теряют возможность его редактирования.
- Очевидным недостатком такой модели является то, что файлы могут оказаться надолго заблокированными, что приводит к простоям в разработке проекта.

2. Копирование — изменение — слияние.

- В данной модели каждый разработчик свободно редактирует свою локальную копию файлов, после чего выполняется слияние изменений.
- Недостаток этой модели в том, что может возникать необходимость разрешения конфликтов между изменениями файла.

В случае слияния, инженер, сохраняющий свою копию вторым, будет вынужден решить конфликты вручную и сохранить эти изменения.



Более подробно о системах контроля версий можно почитать тут :

https://ru.hexlet.io/courses/git_base/lessons/vcs_intro/theory_unit

<http://www.amse.ru/courses/cpp1/2010.02.10.html>

http://all-ht.ru/inf/prog/p_0_0.html

Мы рассмотрели пример работы с SVN, так как он проще для понимания. Наиболее популярными на сегодняшний день являются распределенные системы контроля версий, в частности GIT. Подробно почитать о нем можно тут:

<https://git-scm.com/book/ru/v1/>

Теперь стоит вспомнить о баг-репорте и полях **Affected version/Fixed version** и разобраться что это за версии.

При каждом сохранении разработчиками кода в репозиторий, мы получаем более новую версию кода. Допустим вы работаете с продуктом, собранным из версии 2.250. Вы находите в нем дефект. Это значит что Affected version будет 2.250. Мы не можем знать, есть ли этот дефект в других версиях, возможно, он был внесен последними изменениями - и в версии 2.249 его еще нету. Если есть необходимость, мы всегда можем взять более старую версию и проверить. Для облегчения задачи разработчику по устранению дефекта полезно указать версию в которой дефекта еще точно не было, если мы о такой знаем. Допустим, мы проходили этот тест кейс две недели назад на версии 2.198 и функционал работал правильно. Значит дефект был внесен одним из изменений между версией 2.198 и 2.250.

Далее разработчик исправляет дефект, и при сохранении изменений в репозиторий, версия ПО получается 2.263. Таким образом во всех сборках с номером версии 2.263 и больше, дефект должен быть исправлен.

3.2. Сборка и развертывание. (Build and deploy)

Выше уже не раз упоминалось понятие сборки проекта. Сборка (build) - это процесс преобразования исходного кода, написанного программистами (тот самый код, который хранится в репозитории) в конечный работающий продукт. Сборка включает в себя несколько этапов, которые могут отличаться в зависимости от используемого языка программирования, технологий, архитектуры и т.д.

Основными этапами являются:

1. Компиляция - файлы исходного кода преобразуются в промежуточный код или даже в выполняемый код - для простых программ.
2. Связывание (для более сложных программ) - (нахождение реального положения всех функций, обозначенных как внешние). Это выполняется специальной программой - Линкером. Процесс линковки представляет собой замену

относительных адресов функций внешних библиотек на реальные адреса которые будут использоваться программой в процессе ее выполнения.

Со сборкой связано понятие версии приложения. Линкер часто может автоматически устанавливать (увеличивать) [номер версии](#). Обратите внимание, версияность сборок будет отличаться от версии исходного кода в репозитории. Ведь мы не собираем каждую-каждую версию.

Итак, **сборка** (от англ. *to build* - сооружать, строить) - конечный результат компиляции программы с уникальным номером версии сборки.

После этапа сборки необходимо “развернуть продукт” (**to deploy**) на окружении, где он будет использоваться.

Любой из нас занимался развертыванием программных продуктов. Когда? При установке любого приложения для Windows с помощью exe-файла инсталлятора. Инсталлятор в данном случае - это программа, которая выполняет автоматическое развертывание программы (простите за тавтологию) в нашей среде - копирует нужные файлы, прописывает переменные окружения и т.д. В тестировании есть раздел, посвященный тестированию инсталляции, так как это очень важная часть любого программного продукта.

Если же речь идет о развертывании клиент-серверных продуктов - то это требует большего количества операций, чем просто запуск программы-инсталлятора:

1. Подключиться к целевому серверу.
2. Залить обновление кода из репозитория.
3. Выполнить предписанные инструкции (перезапуск демонов, сброс индексов, обновление структуры БД и прочее).
4. И т.д.

Для упрощения процесса деплоя существует ряд инструментов.

Подробнее можно почитать тут :

https://en.wikipedia.org/wiki/Software_deployment

Простая аналогия для понимания процесса сборки и развертывания - допустим мы собираем двигатель (это процесс build). Он уже собран и может работать. Но сам по себе он нам не принесет пользы. Его необходимо установить правильным образом в автомобиль - это уже процесс развертывания (deploy). И только когда двигатель собран и установлен корректно - автомобиль сможет поехать (при условии корректной работы всех остальных систем - так же, как и с ПО).

3.3. Непрерывная интеграция (CI, *Continuous Integration*)

Теперь мы видим, что от момента написания кода до его работы есть много этапов, на каждом из которых потенциально могут быть ошибки.

Допустим, несколько разработчиков работают над кодом на протяжении недели. Спустя неделю они начинают заливать свой код в репозиторий. Первая проблема с которой они столкнутся - мердж-конфликты. После их решения следующей проблемой будет сборка ПО (скорей всего с первого раза сборка не пройдет). После решения проблем со сборкой могут возникнуть проблемы при развертывании ПО. Ну и если уже наконец-то развернули и запустили - тестировщики наверняка найдут дефекты, регрессию и т.д. Допустим, одно из изменений поломало работающий ранее важный функционал. Среди всех изменений за неделю будет непросто найти, какое именно внесло регрессию. Чем реже происходит процесс сборки, развертывания и тестирования - тем больше сложностей возникает, больше времени уходит на поиск причины и устранение ошибок. Становится очевидным, что чем чаще мы будем собирать, разворачивать и тестировать продукт - тем меньше времени будет уходить на устранение ошибок и неполадок, поиск причин поломок и их исправление.

Согласно [Wikipedia](#) термин **Continuous Integration** введен Мартином Фаулером (Martin Fowler) и Кентом Беком (Kent Beck). Данный термин был придуман ими для обозначения практики частой сборки (интеграции) проекта. Максимально частая сборка является логичным продолжением цепочки

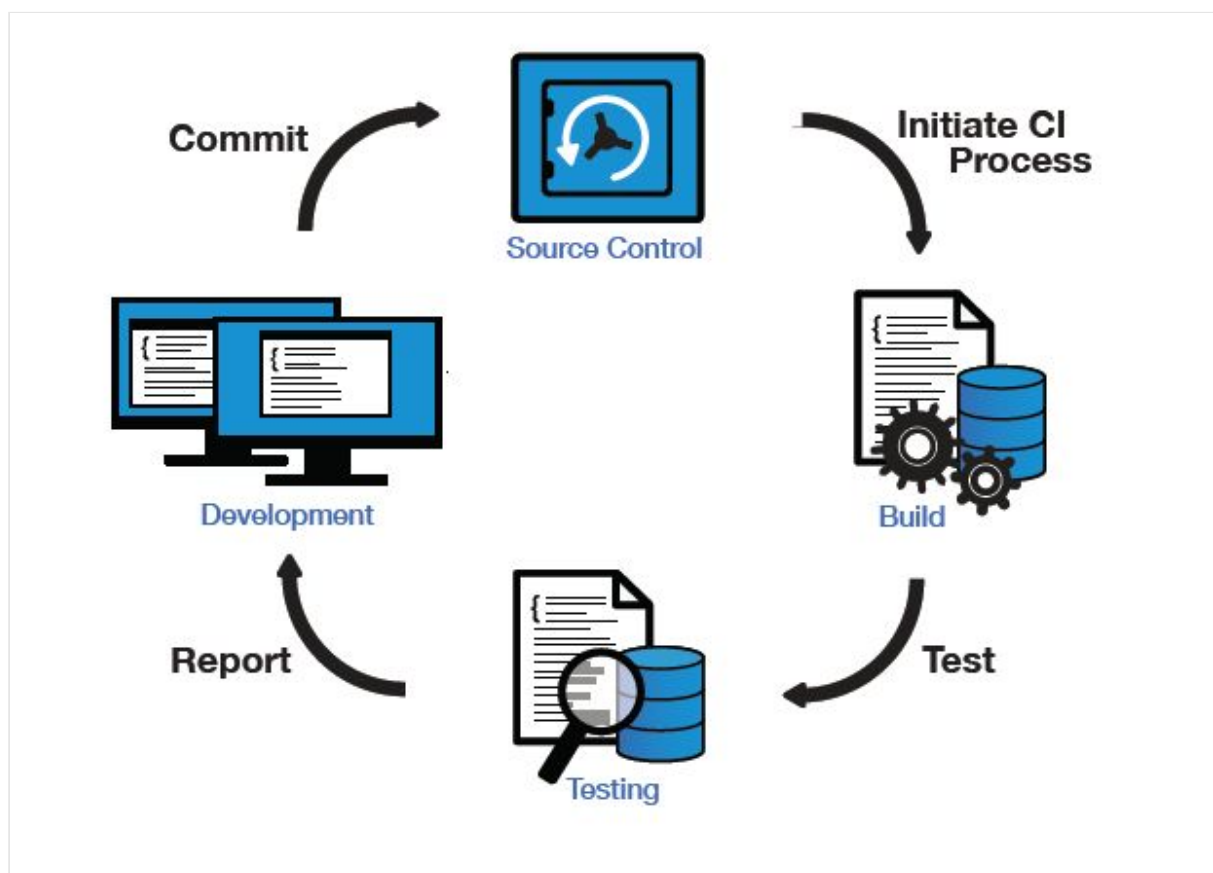
итерационные сборки -> ночные сборки -> непрерывная сборка

В настоящее время Continuous Integration (непрерывная интеграция) одна из практик применяемых в семействе гибких (Agile) методологий. В подобных методологиях она

удачно сочетается с другими практиками, такими как модульное (unit) тестирование, рефакторинг, стандарт кодирования. Но даже без них можно получить пользу от непрерывной интеграции.

Непрерывная интеграция (CI) — это практика разработки программного обеспечения, которая заключается в выполнении частых автоматизированных сборок проекта для скорейшего выявления и решения интеграционных проблем. В обычном проекте, где над разными частями системы разработчики трудятся независимо, стадия интеграции является заключительной. Она может непредсказуемо задержать окончание работ. Переход к непрерывной интеграции позволяет снизить трудоемкость интеграции и сделать её более предсказуемой за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Выглядеть это должно примерно так:



Посмотрим на CI методом белого ящика. Нужно понимать, что система непрерывной интеграции состоит из множества подсистем. Во-первых, нам нужна система контроля

версий (Git, HG и т.д.) из которой будут забираться исходники. Также нам нужен билд-скрипт, который компилирует код и нужны скрипты для развертывания базы данных. Помимо этого нам нужен сервис по запуску сборки, запуску тестов. В зависимости от желания, необходимости и возможностей инструментария можно прикрутить множество рюшечек, например, статический анализ кода, анализ покрытия кода тестами и другие. Таким образом, базовый процесс интеграции выглядит следующим образом:

- Триггер. Событие, при котором запускается сборка продукта. Таким событием может быть: изменения в коде (push), определенное время, нажатие на кнопку.
- После срабатывания триггера стартует сборка проекта из исходников.
- Развертывание базы данных.
- Развертывание приложения.
- Тесты. Авто-тесты не являются обязательными, но их выполнение крайне желательно. Это один из важных пунктов хороших практик CI. К тестам мы еще вернемся.
- Статус, отчеты, уведомления по результатам сборки. После прогона тестов получаем результат сборки, детальные отчеты по каждому из этапов интеграции

Очень здорово, если вы добились стабильной сборки системы. Но пользы будет гораздо больше, если будут применены лучшие практики CI. Про узкие места непрерывной интеграции я расскажу чуть позже.

Преимущества. Какие плюсы от Continuous Integration мы получаем:

- Прежде всего, это регулярная интеграция всего приложения.
- Все делается автоматически, люди избавлены от рутины.
- Экономия время.
- Работа над проектом прозрачна для всех участников команды. Становится проще ответить на вопросы что? где? когда?
- Уменьшаются риски получить «гранату». Дефекты находятся раньше. Это достигается путем запуска тестов (тесты можно запускать различных уровней: unit, GUI, API) и ранней отдачи нового/измененного функционала на тестирование.

- У нас есть всегда (ладно, практически всегда) развернутое окружение для тестирования и демонстрации работы заказчику и прочим заинтересованным. Если ваша команда большая, и вы работаете одновременно в разных ветках репозитория. То теперь буквально в несколько движений вы можете настроить ветку кода на нужное окружение или собрать новое с нужной веткой.
- Можно безболезненно эмитировать процесс деплоя на тестовых серверах.
- Хорошая CI система позволяет поддерживать ряд инженерных практик (анализ кода, покрытие кода, юнит-тесты).

Основным недостатком непрерывной интеграции является трудоемкость поддержки окружения.

Самыми популярными системами непрерывной интеграции на сегодняшний день являются [CruiseControl](#) , [TeamCity](#), [Hudson](#) и [Jenkins](#).

Ссылки : https://en.wikipedia.org/wiki/Continuous_integration

http://devopswiki.net/index.php/%D0%9D%D0%B5%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%BD%D0%B0%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F

http://lib.custis.ru/%D0%9D%D0%B5%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%BD%D0%B0%D1%8F_%D0%B8%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D1%86%D0%B8%D1%8F

https://habrahabr.ru/company/icl_services/blog/262173/

<http://eax.me/jenkins/>

<http://bugscatcher.net/archives/2810>

<https://www.ibm.com/developerworks/ru/library/d-continuous-delivery-framework-jenkins/>

3.4. Техники тест дизайна

Методы (или техники) разработки тестов (test design) – это способ создания тестов. Техники содержат теоретическую часть (некоторые рекомендации по использованию), но главная их часть – практическая. То есть, каждая техника дает нам советы по применению, но как мы будем ее использовать – зависит от нас. Поэтому особенно важно не только изучить технику, но и попробовать ее на практике. Техники могут содержать рекомендации не только по тест дизайну (разработке тестов), но и по выполнению тестов.

Техники тест дизайна - это набор подходов, которые позволяют создать минимальный набор тестовых данных/случаев, который максимально покрывает тестируемый функционал.

Рассмотрим несколько основных техник тест дизайна:

- Разбиение на классы эквивалентности
- Анализ граничных значений
- Тестирование ортогональных массивов
- Причина/Следствие
- Предугадывание ошибок

Разбиение по классам эквивалентности (Equivalence Class Partitioning) – целью техники является уменьшить количество тест-кейсов и не потерять качество, исключить избыточность в тестировании. Основана эта техника на аксиоме: если тестирование одного значения в эквивалентном классе находит дефект, то тестирование другого любого значения в этом классе тоже найдет дефект, и наоборот. Таким образом мы разбиваем входные данные на диапазоны и допускаем, что протестировать одно значение из диапазона будет достаточно.



Анализ граничных значений (Boundary Value Testing) - помогает выбирать наиболее эффективные значения для тестирования. Эта техника применима на всех уровнях тестирования - unit, integration, system, and system-integration test levels.

Подход:

- Определить диапазон значений (как правило это класс эквивалентности).
- Определите границы диапазонов.
- На каждую границу создать 3 тест кейса - один, проверяющий значение границы, второй на значение ниже границы и третий на значение выше границы.

Тестирование ортогональных массивов (Pair-wise Testing) - это современная и эффективная методика тестирования, основанная на том предположении, что большинство дефектов возникает при взаимодействии не более двух факторов. Тестовые наборы, генерируемые при использовании данной методики, охватывают все уникальные пары комбинаций факторов, что считается достаточным для обнаружения большего числа дефектов. Принцип pairwise в том, что в подавляющем большинстве случаев нам не надо проводить полнофакторный эксперимент (т.е. перебирать все конфигурации, где все значения всех параметров встречаются друг с другом). Да и невозможно это зачастую из-за нехватки ресурсов.

Поэтому декларируется, что достаточно проверить как ПО работает, когда каждое значение каждого параметра встретилось с другим значением каждого другого параметра хотя бы раз. Есть случаи, когда параметры ортогональны и нам незначит их "скрещивать" между друг-другом. За счет этого идет дополнительное уменьшение тестовых конфигураций.

Например есть два браузера: Opera и FF. Есть три ОС: Mac, Win и Lin. Это 6 комбинаций. Допустим, у нас сайт на двух языках (RU и EN). Для полного эксперимента, мы должны те 6 умножить на 2, т.е. каждую из предыдущих конфигураций проверить с обоими языками.

Case	OS	Browser	Lang
1	Win	FF	Ru
2	Win	FF	En
3	Win	Opera	Ru
4	Win	Opera	En
5	Lin	FF	Ru
6	Lin	FF	En
7	Lin	Opera	Ru
8	Lin	Opera	En
9	Mac	FF	Ru
10	Mac	FF	En
11	Mac	Opera	Ru
12	Mac	Opera	En

Но зачем? Мы можем воспользоваться pairwise подходом и вместо 12 конфигураций получить 6. Этот метод заставляет задуматься, а важно ли нам проверять каждую ОС в сочетании с другими параметрами. Очевидно - нет, важно например, посмотреть, как каждая ОС работает с каждым языком. Условия задачи соблюдены - мы планируем протестировать каждый браузер, на каждой платформе; каждый язык сайта с каждым браузером и на каждой платформе. Что еще нужно для тестирования? Идеальный вариант, когда каждый параметр с каждым встретится только однажды, возможен только если у всех параметров одинаковое количество значений и равно количеству параметров.

Причина / Следствие (Cause/Effect – CE).

Диаграммы причинно-следственных связей — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий. Причиной в данном случае называют отдельное входное условие или класс эквивалентности. Следствием - выходное условие или преобразование системы.



Например, вы проверяете возможность добавлять клиента, используя определенную экранную форму. Для этого вам необходимо будет ввести несколько полей, таких как "Имя", "Адрес", "Номер Телефона" а затем, нажать кнопку "Добавить" - эта "Причина". После нажатия кнопки "Добавить", система добавляет клиента в базу данных и показывает его номер на экране - это "Следствие". Данный метод позволяет строить высокорезультативные тесты и обнаруживать неполноту и неоднозначность исходных требований.

Предугадывание ошибки (Error Guessing - EG)

Это техника базируется на том, что аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать", при каких входных условиях система может выдать ошибку. Например, спецификация

говорит: "пользователь должен ввести код". Тест аналитик, будет думать: "Что, если я не введу код?", "Что, если я введу неправильный код? ", и так далее. Это и есть предугадывание ошибки. С опытом, приходит понимание типичных узких мест в продукте, что позволяет эффективно предугадывать ошибки.

УРОК 4.

4.1. Понятие дефект

Дефект (Defect, Bug) — отклонение фактического результата (actual result) от ожиданий наблюдателя (expected result), сформированных на основе требований, спецификаций, иной документации или опыта и здравого смысла.

Ожидаемый результат — поведение системы, описанное в требованиях.

Фактический результат — поведение системы, наблюдаемое в процессе тестирования.

Дефекты бывают разных классификаций в зависимости от вида тестирования. Например, функциональное тестирование, тестирование документации, тестирование производительности и т.п. Отсюда логически вытекает, что дефекты могут встречаться не только в коде приложения, но и в любой документации, в архитектуре и дизайне, в настройках тестируемого приложения или тестового окружения — где угодно.

Синонимы: ошибка, отклонение, недочет, аномалия, помеха, отказ, проблема.

Итак, как только мы обнаруживаем баг, нам необходимо его задокументировать для продолжения жизненного цикла дефекта (который мы рассматривали ранее).

Документ, который описывает баг, называется – баг репорт.

Баг репорт (bug report) – это технический документ, который содержит в себе полное описание бага, включающее информацию, как о самом баге (короткое описание, серьезность, приоритет и т.д.), так и о условиях возникновения данного бага. Баг репорт должен содержать правильную, единую терминологию, описывающую элементы пользовательского интерфейса и события данных элементов, приводящих к возникновению бага.

В общем случае, баг репорт **состоит из:**

Шапка:

- Короткое описание (короткое описание проблемы)
- Проект (название текущего проекта)
- Компонент приложения (в котором возник дефект)
- Версия (версия билда, в котором найден баг)
- Серьезность (градация степени влияния на приложение бага)
- Приоритет (очередь исправления бага)
- Статус (отображает статус бага в своем жизненном цикле)
- Автор (автор баг репорта)
- Назначение (кто должен исправить дефект)

Окружение:

- Операционная система, разрядность, пакет обновления (service pack, сокращенно SP), браузер, его версия и т.д.

Описание:

- Шаги воспроизведения (описание пути, который приводит к возникновению дефекта)
- Фактический результат (результат, к которому приходим, выполнив все шаги воспроизведения)
- Ожидаемый результат (результат, который должен быть (соответственно требованиям)).

Дополнения:

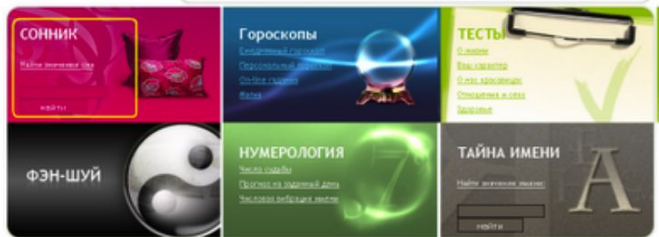
- Прикрепленный файл (логи, скриншоты, другие документы, которые могут помочь воспроизвести проблему или решить ее)

Несмотря на такое большое количество пунктов баг репорта, можно выделить несколько **основных полей**, присутствие которых необходимо:

- **Краткое описание.** Поле, в котором нужно поместить весь смысл всего баг репорта. Чаще всего, в коротком описании лаконично отвечают на 3 вопроса: «Где?», «Что?», «Когда?» (именно в такой последовательности, как бы не хотелось изменить ее по примеру всем известной игры).

- **Серьезность.** Дефект либо полностью останавливает работоспособность приложения, либо только часть функциональности, либо иное.
- **Шаги к воспроизведению.** Точное и понятное описание всех шагов, которые приводят к появлению дефекта, с учетом всех необходимых входных данных и т.д.
- **Фактический результат.**
- **Ожидаемый результат.**

Рекомендованная ссылка: <http://www.protesting.ru/testing/bugwriting.html>

Короткое описание	Поиск в соннике на главной странице, с использованием русских слов, работает не правильно.
Проект	http://www.ameno.ru/
Компонент приложения	Поиск в соннике
Номер версии	0.001
Важность: <ul style="list-style-type: none"> • S1 Блокирующая (Blocker) • S2 Критическая (Critical) • S3 Значительная (Major) • S4 Незначительная (Minor) • S5 Тривиальная (Trivial) 	S3 Значительная (Major)
Приоритет: <ul style="list-style-type: none"> • P1 Высокий (High) • P2 Средний (Medium) • P3 Низкий (Low) 	заполняется менеджером
Статус	Новая
Автор	Алексей Булат
Назначен на	имя разработчика
Шаги воспроизведения	<p>1. Открываем главную страницу сайта: http://www.ameno.ru/ --> Внизу страницы находим раздел: СОННИК (см. копию экрана - выделено желтой рамкой)</p>  <p>3. Введите поисковое слово, например "ночь" 4. Нажмите кнопку "Найти"</p>

Жизненный цикл дефектов

Итак, мы нашли баг. Может даже блокер. Что же с ним может случиться на всём его нелегком жизненном пути? (Названия этапов жизни дефектов могут быть разными в разных баг-трекинг системах, но суть их одна).

- **Новый (New).** Тестировщик нашел баг, дефект успешно занесен в «Bug-tracking» систему.
- **Открыт (Opened).** После того, как тестировщик отправил ошибку, она либо автоматически, либо вручную назначается на человека который должен её проанализировать (обычно Project Manager). В зависимости от решения менеджера проекта, баг может быть:
 - **Отложен (Deferred).** Исправление этого бага не несет ценности на данном этапе разработки или по другим, отсрочивающим его исправление причинам.
 - **Отклонен (Rejected).** По разным причинам дефект может и не считаться дефектом или считаться неактуальным дефектом, что вынуждает отклонить его.
 - **Дубликат (Duplicate).** Если описанная ошибка уже ранее была внесена в «Bug-tracking» систему, то статус такой ошибки меняется на «дубликат».
 - **Назначен (Assigned).** Если ошибка актуальна и должна быть исправлена в следующей сборке (build), происходит назначение на разработчика который должен исправить ошибку.

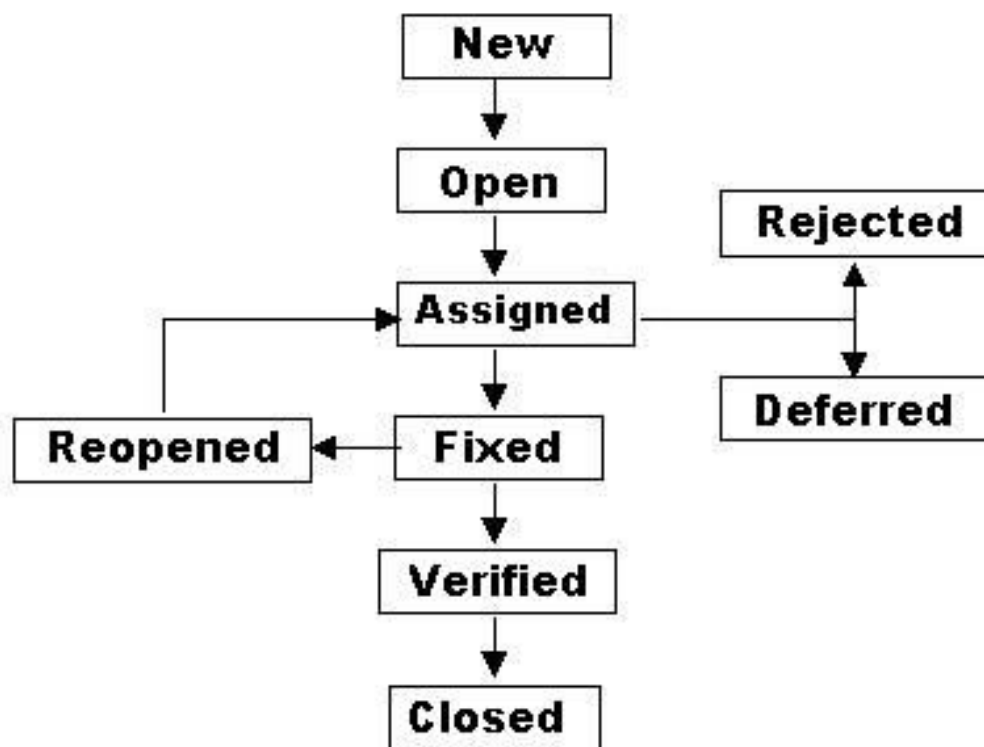
Когда наличие дефекта неопровержимо, его путь может привести к следующим статусам:

- **Исправлен (Fixed).** Ответственный за исправление бага разработчик заявляет, что устранил дефект.

В зависимости от того, исправил ли разработчик дефект, дефект может быть:

- **Проверен (Verified).** Тестировщик проверяет, действительно ли ответственный разработчик исправил дефект, или все-таки разработчик безответственный. Если бага больше нет, он получает данный статус.

- **Повторно открыт (Reopened).** Если опасения тестировщика оправданы и баг в новом билде не исправлен, – он все так-же потребует исправления, поэтому заново открывается.
- **Закрытый (Closed).** В результате определенного количества циклов баг все-таки окончательно устранен и больше не потребует внимания команды – он объявляется закрытым.



Классификация дефектов

Классификация дефектов, с точки зрения **степени влияния (Severity)** на работоспособность ПО:

- **Blocker.** Ошибка, которая приводит программу в нерабочее состояние. Дальнейшая работа с программной системой или ее функциями невозможна.

- **Critical.** Критический дефект, приводящий некоторый ключевой функционал в нерабочее состояние. Также это может быть существенное отклонение от бизнес логики, неправильная реализация требуемых функций, потеря пользовательских данных и т.д.
- **Major.** Весьма серьезная ошибка, свидетельствующая об отклонении от бизнес логики или нарушающая работу программы. Не имеет критического воздействия на приложение.
- **Minor.** Незначительный дефект, не нарушающий функционал тестируемого приложения, но который является несоответствием ожидаемому результату. Например, ошибка дизайна.
- **Trivial.** Баг, не имеющий влияние на функционал или работу программы, но который может быть обнаружен визуально. Например, ошибка в тексте.

Градация дефектов, с точки зрения **приоритетности исправления (Priority)**:

- **High.** Баг должен быть исправлен как можно быстрее, т.к. он критически влияет на работоспособность программы.
- **Medium.** Дефект должен быть обязательно исправлен, но он не оказывает критическое воздействие на работу приложения.
- **Low.** Ошибка должна быть исправлена, но она не имеет критического влияния на программу и устранение может быть отложено, в зависимости от наличия других, более приоритетных дефектов.

С помощью такой классификации организована работа многих систем отслеживания ошибок, в том числе Jira. Ознакомившись с этими терминами, Вы сможете лучше понимать жизненный цикл дефекта, а это необходимый шаг, на пути становления тестировщика.

4.2. JIRA

JIRA - это продукт, предназначенный для организации процесса контроля запросов и задач, имеющий часть функциональности, обычно присущей большим и дорогим системам управления проектами.

Что такое JIRA

JIRA – это инструмент для организации эффективного взаимодействия участников процесса или проекта



учет

- типы запросов
- атрибуты
- файлы
- связи и поиск
- уровни доступа



процессы

- этапы процесса
- ответственные
- уведомления
- маршруты
- нормативы



общение

- рабочие столы
- комментарии
- лента событий
- история
- подписка



аналитик

- учет трудозатрат
- отчеты по стадиям
- тренды
- анализ активности
- выгрузка в Excel

Концепция JIRA

► Проекты (Projects)

- Позволяют группировать задачи для отдельных проектов и групп пользователей
- Имеют свои правила безопасности, интерфейсов, уведомлений и процессов
- Позволяют группировать задачи в продукты и компоненты проекта
- Есть возможность обеспечить многоуровневый доступ к задачам одного проекта

► Задачи/Запросы (Tasks/Issues)

- Могут быть различного типа с уникальными атрибутами
- За ходом исполнения задач могут наблюдать заинтересованные сотрудники
- Есть атрибутивный поиск задач
- Планирование и учет трудозатрат по решения задач
- Хранение в задаче любых файлов

► Подзадачи (Sub-Tasks)

- Позволяют разбивать задачи на самостоятельные этапы

Jira - инструментальное средство управления отчетами о дефектах.

Так называемые «инструментальные средства управления отчетами о дефектах» в обычной разговорной речи называют «баг-трекинговыми системами», «баг-трекерами».

Как правило, Jira покрывает стандартные функции:

- создание отчетов о дефектах, управление их жизненным циклом с учетом контроля версий, прав доступа и разрешённых переходов из состояния в состояние
- сбор, анализ и предоставление статистики в удобной для восприятия человеком форме
- рассылка уведомлений, напоминаний и иных артефактов соответствующим сотрудникам
- организация взаимосвязей между отчетами о дефектах, тест-кейсами, требованиями и анализ таких связей с возможностью формирования рекомендаций
- подготовка информации для включения в отчет о результатах тестирования
- интеграция с системами управления проектами.

Иными словами, хорошее инструментальное средство управления жизненным циклом отчетов о дефектах не только избавляет человека от необходимости внимательно выполнять большое количество рутинных операций, но и предоставляет дополнительные возможности, облегчающие работу тестировщика и делающие ее более эффективной.

УРОК 5.

5.1. Мобильное тестирование

Поскольку все больше и больше программного обеспечения разрабатывается под мобильные устройства, стоит отдельно рассмотреть особенности тестирования мобильных приложений.

Специфика работы программного обеспечения на мобильном устройстве накладывает определенные требования и ограничения - начиная от потребления батареи и

заканчивая взаимодействием с другими приложениями. Еще одной проблемой является наличие огромного количества устройств с разными характеристиками - размерами экрана, ресурсами, операционными системами.

Протестировать приложение на всех возможных устройствах и исправить найденные дефекты практически невозможно (как минимум это очень долго и дорого) поэтому производители мобильных приложений классифицируют устройства на более приоритетные и менее приоритетные. Чаще всего используют следующие обозначения:

Gold(White) - устройства , которые поддерживаются полностью

Silver(Grey) - устройства, которые поддерживаются частично (устраняются в основном серьезные дефекты, которые затрагивают непосредственно функционал)

Black - устройства, которые не поддерживаются.

Классификация базируется на статистических данных пользователей и популярности устройств.

Рассмотрим подробнее какие проверки необходимы при тестировании мобильных приложений.

Их можно поделить на следующие разделы:

1. Характеристики устройства. Они относятся к устройству, на которое устанавливается приложение.
2. Сетевые характеристики.
3. Проверки приложения. Они относятся к функциональности, которая часто используется.
4. Проверки интерфейса
5. Проверки, специфичные для магазина приложений

Ниже приведен довольно полный чеклист.

1. ХАРАКТЕРИСТИКИ УСТРОЙСТВА

1.1 Можно ли установить приложение?

- 1.2 Ведет ли оно себя правильно при входящем звонке?
- 1.3 Ведет ли оно себя правильно при входящем SMS?
- 1.4 Ведет ли оно себя правильно при подключении зарядного устройства?
- 1.5 Ведет ли оно себя правильно при отключении зарядного устройства?
- 1.6 Ведет ли оно себя правильно, если устройство переведено в спящий режим?
- 1.7 Ведет ли оно себя правильно, если устройство выведено из спящего режима?
- 1.8 Ведет ли оно себя правильно при разблокировке экрана?
- 1.9 Ведет ли оно себя правильно при повороте устройства?
- 1.10 Ведет ли оно себя правильно при встряхивании устройства?
- 1.11 Ведет ли оно себя правильно при сообщении от другого приложения (напоминания календаря, список задач, и т. п.)?
- 1.12 Ведет ли оно себя правильно при получении пуш-сообщений от другого приложения (упоминания в твиттере, сообщения WhatsApp, и т. п.)?
- 1.13 Правильно ли оно взаимодействует с GPS- сенсором (при его включении/выключении, использовании геолокационных данных)?
- 1.14 Определена ли для него функциональность всех кнопок/клавиш устройства?
- 1.15 Убедиться, что кнопки/клавиши, не ассоциированные с функциями приложения, не вызывают неожиданного поведения при активации.
- 1.16 Если на устройстве доступна физическая кнопка "назад", переводит ли она пользователя на предыдущий экран?
- 1.17 Если на устройстве доступна физическая кнопка "меню", открывает ли она меню приложения?
- 1.18 Если на устройстве доступна физическая кнопка "домой", переносит ли она пользователя на домашний экран, если приложение запущено?
- 1.19 Если на устройстве доступна физическая кнопка "поиск", открывает ли она поиск внутри приложения?
- 1.20 Ведет ли приложение себя правильно при сообщении о недостаточном заряде батареи?
- 1.21 Ведет ли оно себя правильно, если на устройстве выключен звук?
- 1.22 Ведет ли оно себя правильно, если устройство находится в режиме "авиа"?
- 1.23 Можно ли деинсталлировать приложение?

- 1.24 Ведет ли оно себя правильно после переустановки?
- 1.25 Доступно ли оно при поиске в магазине приложений (проверять после одобрения приложения в магазине)?
- 1.26 Может ли приложение переключаться на другие приложения устройства через режим мультитасчности (если должно)?

2. СЕТЕВЫЕ ХАРАКТЕРИСТИКИ

- 2.1 Соответствует ли поведение приложения желаемому, если оно подключено к Интернету через Wi-Fi?
- 2.2 Соответствует ли поведение приложения желаемому, если оно подключено к Интернету через 3G?
- 2.3 Соответствует ли поведение приложения желаемому, если оно подключено к Интернету через 2G?
- 2.4 Соответствует ли поведение приложения желаемому, если сеть недоступна?
- 2.5 Возобновляет ли приложение работу, когда снова получает доступ к сети после прерванного доступа?
- 2.6 Обновление транзакций проходит корректно после переподключения к сети.
- 2.7 Продолжает ли приложение корректно работать, если оно привязано или каким-либо другим образом соединено с другим устройством?
- 2.8 Что происходит, если приложение переключается между сетями (Wi-Fi, 3G, 2G)?
- 2.9 Использует ли приложение стандартные сетевые порты (Почта: 25, 143, 465, 993 или 995, HTTP: 80 или 443, SFTP: 22) для удаленных подключений: некоторые провайдеры блокируют отдельные порты.

3. ХАРАКТЕРИСТИКИ ПРИЛОЖЕНИЯ

- 3.1 Тестировалось ли оно на различных устройствах/версиях ОС?
- 3.2 Проверка стабильности: если в приложении есть списки (например, изображений), попробуйте быстро их пролистать.

- 3.3 Проверка стабильности: если в приложении есть списки (например, изображений), попробуйте пролистать их до позиции "до первого изображения" и "после последнего".
- 3.4 Прекращается ли загрузка приложения, если оно превышает допустимый в ОС размер для загрузки через мобильный интернет?
- 3.5 Интеграция: правильно ли оно подключается к соцсетям (LinkedIn, Twitter, Facebook, и т. п.).
- 3.6 Приложение не вмешивается в работу других приложений в фоновом режиме (GPS, проигрывание музыки, и т. п.)
- 3.7 Можно ли печатать из приложения (если применимо)
- 3.8 Функциональность поиска отображает релевантные результаты.
- 3.9 Работа распространенных жестов при управлении приложением.
- 3.10 Что произойдет при выборе нескольких опций одновременно (незапланированный мультитач – например, выбор нескольких контактов из записной книжки разом).
- 3.11 Имя приложения должно быть "говорящим" (коротким, но отображающим суть приложения, привлекающим внимание покупателей)
- 3.12 Ограничивает ли приложение кэш, чистит ли его?
- 3.13 Перезагрузка данных от удаленного сервиса сконфигурирована так, чтобы не вызывать проблем с производительностью на стороне сервера (ручная перезагрузка может снизить количество обращений к серверу).
- 3.14 Переходит ли приложение в спящий режим, когда работает в фоне (это экономит заряд батареи)?

4. ПРОВЕРКИ ИНТЕРФЕЙСА

- 4.1 Контролирующие элементы максимально ненавязчивы (к примеру, исчезают, если не используются длительное время).
- 4.1 Пользователь может вернуться на предыдущий экран (например, нажав "Назад" или "Отмена").
- 4.2 Основная функция приложения понятна и очевидна.
- 4.3 Функция, которую, скорее всего, будет использовать пользователь, подсвечена (к примеру, в iOS голубая кнопка обозначает действие по умолчанию/наиболее вероятное действие).

- 4.4 Минимизируйте действия пользователя, используя выпадающие списки или табличное представление, где пользователь может сделать выбор, вместо ручного ввода данных.
- 4.5 Функция поиска должна быть доступна для длинных списков, которые нужно проматывать.
- 4.6 Для ситуаций с низкой производительностью пользователю показывается иконка прогресса ("Загружается...")
- 4.7 В случае "живой" фильтрации данных при вводе поискового запроса проверьте производительность приложения.
- 4.8 Внешний вид кнопок для стандартных действий привычен пользователю (к примеру, обновление, сортировка, корзина, ответ, назад и т. п.)
- 4.9 Стандартные иконки не используются для функций, для которых они в норме не применяются.
- 4.10 Приложение правильно реагирует на смену ориентации устройства.
- 4.11 Не переобозначены жесты, которые выполняют стандартные функции (к примеру, смахивание сверху вниз открывает центр уведомлений).
- 4.12 Требование войти в систему не появляется в приложении, пока не становится необходимым.
- 4.13 Если приложение неожиданно остановлено, пользовательские данные должны быть локально сохранены и быть доступными при старте приложения.
- 4.14 При удалении документов пользователь предупреждается о последствиях.
- 4.15 Клавиатура подстраивается под ожидаемый ввод (к примеру, цифры/буквы).
- 4.16 Легко ли отличить неактивные кнопки от активных?

5. ПРОВЕРКИ, СПЕЦИФИЧНЫЕ ДЛЯ МАГАЗИНА ПРИЛОЖЕНИЙ

Они в основном базируются на гайдлайнах Apple AppStore. Другой широко распространенный магазин приложений – Google Play – куда менее требователен.

- 5.1 Приложение не может использовать "непубличные API". Это означает, что вы не можете использовать некоторые функции, которые платформа использует для своих собственных приложений. Обычно это можно проверить автоматически, например, при помощи <http://www.chimpstudios.com/appscanner/>)

5.2 Приложение не может перепрограммировать контрольные клавиши устройства, не предназначенные для подобного использования (к примеру, использовать кнопку громкости для фотосъемки).

5.3 Приложение не должно иметь доступ к файлам, находящимся вне приложения, без разрешения пользователя (к примеру, копировать адресную книгу или получать информацию от других приложений).

5.4 Приложение не должно иметь доступа к папкам и файлам вне директорий контейнера и документов.

5.5 Приложение не может загружать код для установки без согласия пользователя.

5.6 Приложение может обновляться только через магазин приложений.

5.7 После загрузки приложение должно быть рабочим. Оно не должно отключиться через несколько дней.

5.8 Приложение не может быть "опытной", "бета", "демо" или "тест"-версией.

5.9 Названия продуктов Apple должны быть указаны без опечаток (iPhonez – неверно).

5.10 Если приложение использует сеть, оно не пользуется ею через сторонние (не-Apple) браузеры.

5.11 В приложении не должны упоминаться сторонние платформы (к примеру, "Также доступно для Android!").

5.12 Приложение не может использовать устаревшие интерфейсы (например, колесо iPod).

5.13 Многозадачная функциональность приложения может быть доступна только для его исходных целей (то есть для VoIP, проигрывание аудио, определение местоположения, завершение задач, локальные уведомления, и т. п.). Это значит, что в общем случае приложение не может работать в фоновом режиме и должно быть закрыто, если оно не используется.

5.14 В приложении должна быть хоть какая-то функциональность. Оно не может состоять из одной странички и текста, не может быть просто песней, фильмом или книгой – для этого есть другие платформы.

5.15 Функциональность должна соответствовать описанию в магазине приложений.

5.16 В целом приложение должно быть "достойным". Откровенные материалы – секс, насилие, наркотики, алкоголь, табак – не должны в нем демонстрироваться, и оно не должно отзываться об отдельных этнических или религиозных группах уничижительно. Приложение должно быть честным. Его описание должно быть правдивым, и вся функциональность должна работать, как описано. Если приложение дает диагностическую информацию, она должна быть надежной. Это также касается жанра и категории в магазине. Иконки приложения должны соответствовать и подходить ему.

5.17 Приложение не может ограничивать пользователей в выборе геолокации или мобильного оператора.

5.18 Приложение не может рассылать спам, распространять вирусы, или использовать другие платформы Apple (например, Game Center/Push Notifications) с этими целями.

5.19 Приложение не может использовать геолокационные службы без разрешения.

5.20 Все ссылки в коде приложения должны работать.

5.21 Приложение не может использовать местонахождение пользователя без разрешения.

5.22 Геолокационные службы не могут использоваться для автономного контроля транспортных средств или вызова служб скорой помощи.

5.23 Приложение не может использовать уведомления без согласия пользователя.

5.24 Уведомления не должны содержать личных данных.

5.25 Приложение не может распространять личную информацию пользователей (например, ID игрока) через Game Center.

5.26 Рекламные баннеры должны скрываться, если реклама недоступна.

5.27 Приложение должно уважать авторские права Apple и других сторон.

5.28 Механизм встроенных покупок не может использоваться для приобретения товаров и услуг, использующихся вне приложения.

5.29 Механизм встроенных покупок не может использоваться для сбора средств на благотворительность (для этого есть SMS).

5.30 Механизм встроенных покупок не может использоваться для покупки лотерейных билетов напрямую из приложения.

5.31 Приложения, которые поощряют пользователя использовать устройство так, что оно может быть повреждено, не будут одобрены.

5.32 Приложение не может запрашивать личные данные (например, email) пользователя, чтобы функционировать.

Большинство из выше приведенных проверок (кроме раздела 5. Проверки, специфичные для магазина приложений) также применимы и для тестирования мобильных версий веб-сайтов.

Если поддерживаемых устройств много, возникает другая проблема - как за меньшее время протестировать на всех возможных устройствах? Особенно это актуально для тестирования мобильных версий сайтов, так как помимо устройства и версии ОС добавляются еще разные браузеры и их версии.

Тут на помощь приходит разбиение на **классы эквивалентности** по следующим признакам:

1. Размер дисплея/разрешение - можно допустить, что ошибки связанные с масштабирование графических элементов будут одинаковые для устройств, с одинаковым размером дисплея и разрешением.
2. Операционная система - также можно допустить, что на одинаковых операционных системах с одинаковой версией, ошибки связанные с взаимодействием с ОС будут одинаковы.
3. Производитель устройства - ошибки, связанные с взаимодействием с "железом" скорее всего будут одинаковыми на устройствах одного производителя.

Для уменьшения количества комбинаций устройство-операционная система-браузер используется метод ортогональных массивов (pair-wise testing).

Ссылки:

<http://appadvice.com/appnn/2010/09/apples-app-store-review-guidelines-annotated-explained>

Гайдлайны Apple App Store:

<https://developer.apple.com/appstore/resources/approval/guidelines.html>

Гайдлайны хранения данных iOS:

<https://developer.apple.com/icloud/documentation/data-storage/>

Гайдлайны интерфейса Apple:

<https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/>

<http://www.mobileappstesting.com/tag/testing-checklist-for-mobile-application/>

http://www.vietnamesetestingboard.org/zbxe/?document_srl=529839

5.2. Роли

Хорошо сформированная проектная команда подразумевает, что каждый член команды понимают свою роль в формировании успеха проекта. Каждый член команды знает, какой вклад ему необходимо принести в проект, когда ему необходимо это сделать, чем занимаются другие члены проекта, и что представляет собой успех. Столь же важно, чтобы каждый член команды оказывал поддержку остальным, и таким образом гарантировал успех всего проекта. Для начинающего инженера важно понимать, какова область ответственности каждого члена команды и к кому по каким вопросам можно и нужно обращаться.

На разных проектах команды могут быть сформированы по-разному: разработчики с тестировщиками в одной команде, либо же в разных. На проекте может быть или не быть некоторых ролей, например бизнес аналитика или разработчика баз данных (если для продукта база не нужна). В специфических областях могут быть более редкие профессии - например художники или даже композиторы (писать музыку к играм), с которыми также необходимо будет работать и коммуницировать. Ниже приведены основные роли, которые вы можете встретить :

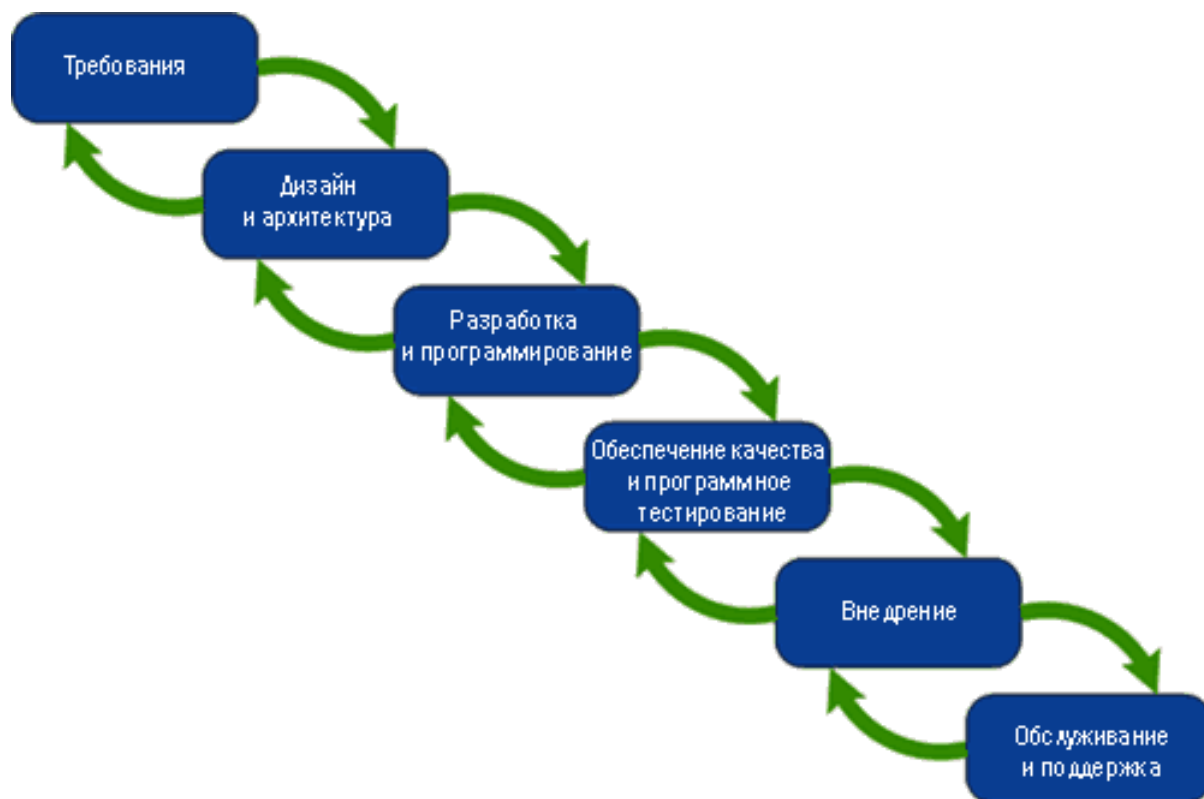
Проектная роль	Описание
Руководитель проекта (Project Manager)	Руководитель проектной команды, ответственный за управление проектом, достижение целей проекта в рамках бюджета, в срок и с заданным уровнем качества. Организует необходимое взаимодействие и способен разрешать проблемы, возникающих по ходу выполнения проекта.
Системный архитектор (SA)	Формирует архитектуру и идеологию проекта, оценку имеющихся технических возможностей, взаимосвязи со смежными системами, проектную документацию.
Бизнес-аналитик (BA)	Обеспечивает сбор требований, их обработку, документирование и передачу разработчикам. Для бизнес-аналитика важно понимать прикладную область и специфику продукта, что необходимо заказчику и как это описать техническим текстом, понятным разработчикам.
Администратор БД (DBA)	Проводит установку и настройку СУБД. Отвечает за выработку требований к БД, ее проектирование, реализацию, оптимизацию, сопровождение и архивирование, включая управление учетными записями пользователей БД и защиту от несанкционированного доступа.
Системный администратор (SysAdmin)	Обеспечивает установку и настройку сервера приложений и его компонентов, разработку дополнительных приложений
Специалист по интеграции разработки и эксплуатации (DevOps - development and operations)	Методология DevOps нацелена на взаимодействие программистов и системных администраторов для увеличения частоты выпуска релизов. Соответственно, DevOps engineer — специалист, который работает на стыке этих двух должностей и занимается автоматизацией жизненного цикла приложения (включая проектирование, разработку, тестирование, развертывание,

	поддержку и мониторинг).
Лидер команды разработчиков (TeamLead)	Организовывает процесс разработки на проекте, распределяет задачи, следит за их выполнением, оценивает эффективность работы команды. Задача лидера - задавать темп и вести команду за собой.
Разработчик приложений (Developer)	Разрабатывает, развивает и сопровождает продукт.
Лидер команды тестировщиков (QA-lead)	Организовывает процесс тестирования на проекте, распределяет задачи, следит за их выполнением, пишет тест планы и оценивает эффективность работы команды.
Инженер по обеспечению качества (QA engineer)	Занимается настройкой процессов, внедрением практик, которые позволят обеспечить качество продукта.
Тестировщик	Проводит контроль качества. Пишет тестовую документацию, разрабатывает тесты, выполняет тестирование продукта.

5.3. Жизненный цикл ПО. Методологии - водопад, V-model, Agile (Scrum)

Тестирование – не изолированный процесс. Это часть модели жизненного цикла программного обеспечения (Software Development Life Cycle, SDLC). Именно поэтому выбор средств и методик тестирования будет напрямую зависеть от выбранной модели разработки. В этом разделе мы рассмотрим наиболее часто применяемые подходы к разработке программного обеспечения, а также популярные сегодня методологии и практики, такие как Agile и Scrum.

Жизненный цикл программного обеспечения (также называемый циклом разработки) – это условная схема, включающая отдельные этапы, которые представляют стадии процесса создания ПО. При этом на каждом этапе выполняются разные действия



Цикл разработки предлагает **шаблон**, использование которого облегчает проектирование, создание и выпуск качественного программного обеспечения. Это методология, определяющая процессы и средства, необходимые для успешного завершения проекта.

Рассмотрим более подробно существующие активности/задачи связанные с тестированием, они привязаны к этапам жизненного цикла ПО.

Стадии цикла разработки ПО

1. Анализ требований - этот этап предполагает сбор требований к разрабатываемому программному обеспечению, их систематизацию, документирование, анализ, а также выявление и разрешение противоречий.

2. Проектирование. На стадии проектирования (называемой также стадией дизайна и архитектуры) программисты и системные архитекторы, руководствуясь требованиями, разрабатывают высокоуровневый дизайн системы.

Разнообразные технические вопросы, возникающие в процессе проектирования, обсуждаются со всеми заинтересованными сторонами, включая заказчика. Определяются технологии, которые будут использоваться в проекте, загрузка команды, ограничения, временные рамки и бюджет. В соответствии с уточненными требованиями выбираются наиболее подходящие проектные решения.

3. Разработка и программирование. После того, как требования и дизайн продукта утверждены, происходит переход к следующей стадии жизненного цикла – непосредственно разработке. Здесь начинается написание программистами кода программы в соответствии с ранее определенными требованиями.

Программирование предполагает четыре основных стадии:

1. Разработка [алгоритмов](#) – фактически, создание логики работы программы;
2. Написание исходного кода;
3. [Компиляция](#) – преобразование в машинный код;
4. Тестирование и отладка – речь, главным образом, о юнит-тестировании.

4. Документация. Этот этап выделяют достаточно условно, поскольку, как мы видели, те или иные документы создаются на всех стадиях жизненного цикла программы. Тем не менее, помимо проектной документации и сопровождающих разработку записей, существуют также и другие текстовые документы, описывающие, например, функции программы и способы ее использования.

5. Тестирование. Основные этапы тестирования мы уже рассматривали ранее, в разделе Фундаментальный процесс тестирования.

Тестировщики занимаются поиском дефектов в программном обеспечении и сравнивают описанное в требованиях поведение системы с реальным. В фазе тестирования обнаруживаются пропущенные при разработке баги. При обнаружении дефекта, тестировщик составляет отчет об ошибке, который передается разработчикам. Последние его исправляют, после чего тестирование повторяется – но на этот раз для того, чтобы убедиться, что проблема была исправлена, и само исправление не стало причиной появления новых дефектов в продукте.

Тестирование повторяется до тех пор, пока не будут достигнуты критерии его окончания.

Виды, методы и техники тестирования мы подробно рассмотрим дальше в этом пособии.

6. Внедрение и сопровождение. Когда программа протестирована и в ней больше не осталось серьезных дефектов, приходит время релиза и передачи ее конечным пользователям.

После выпуска новой версии программы в работу включается отдел технической поддержки. Его сотрудники обеспечивают обратную связь с пользователями, их консультирование и поддержку. В случае обнаружения пользователями тех или иных пост-релизных багов, информация о них передается в виде отчетов об ошибках команде разработки, которая, в зависимости от серьезности проблемы, либо немедленно выпускает исправление (т.н. hot-fix), либо откладывает его до следующей версии программы.

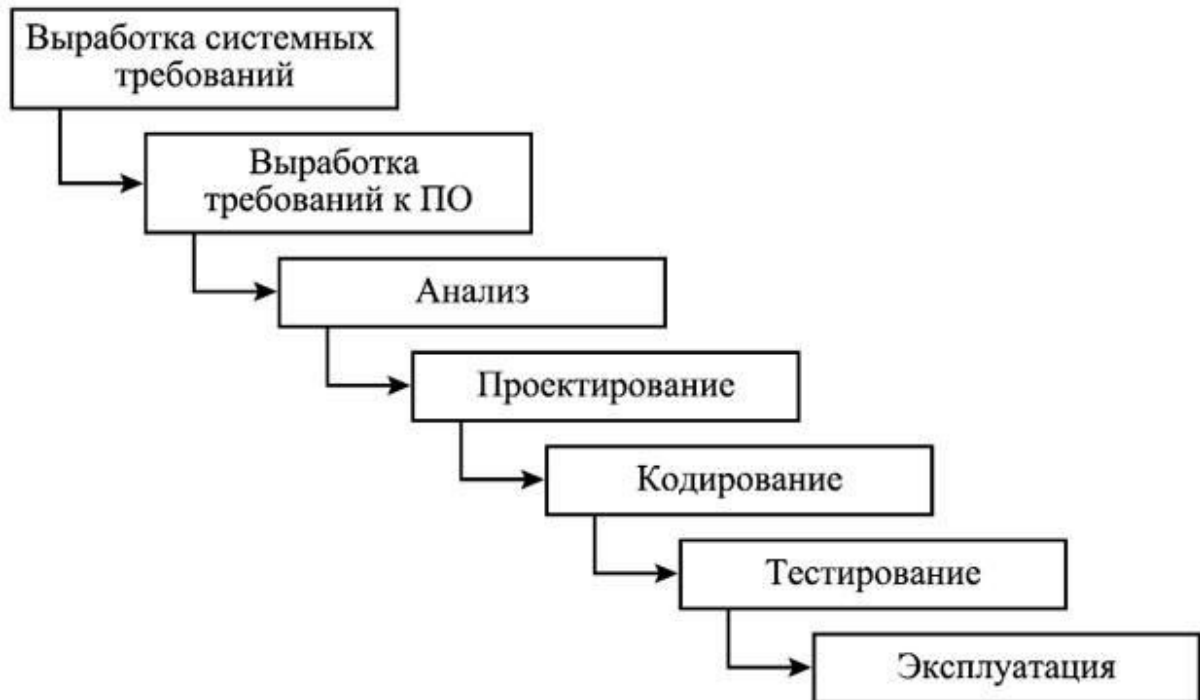
Кроме того, команда технической поддержки помогает собирать и систематизировать различные [метрики](#) – показатели работы программы в реальных условиях.

Заключение:

Все стадии жизненного цикла ПО, представленные выше, **применяются в любой модели разработки**, но их продолжительность и порядок следования могут отличаться. В следующих разделах мы детально рассмотрим основные модели и практики, которые используются современными IT-компаниями в процессе разработки программного обеспечения.

"Водопад" или каскадная модель

Модель предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.



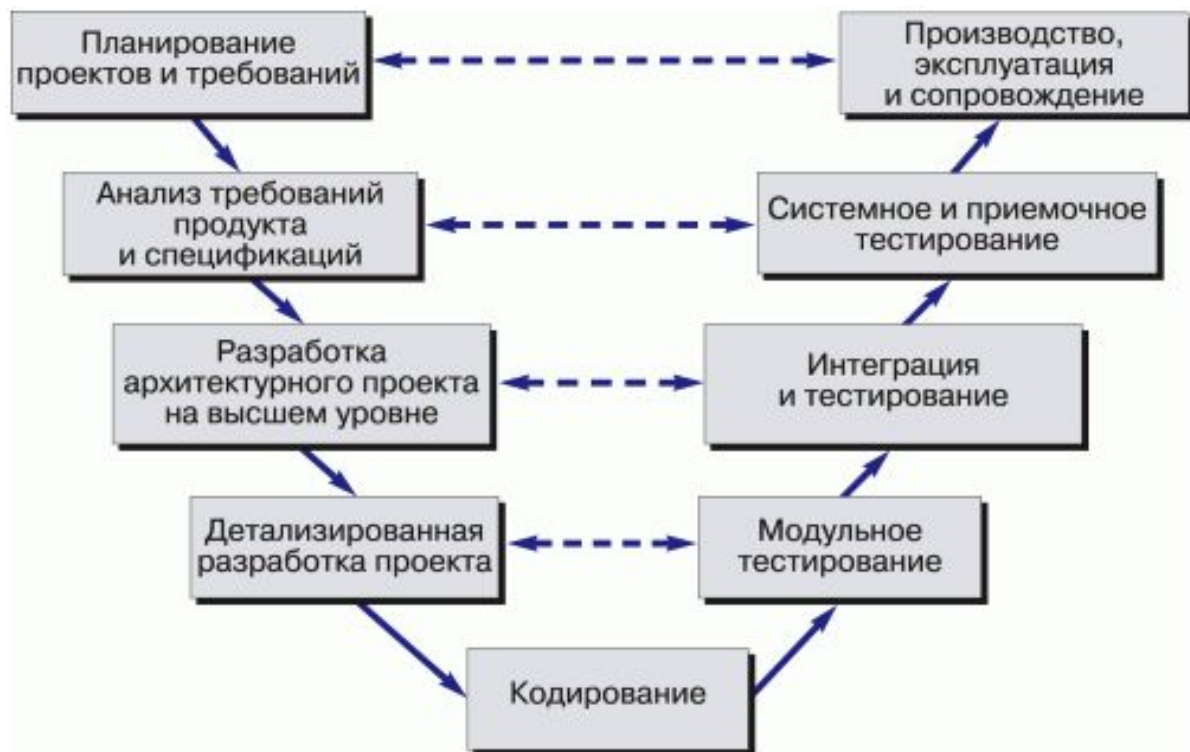
Преимущества:

Последовательное выполнение этапов проекта в строго фиксированном порядке, позволяет оценивать качество продукта на каждом этапе

Недостатки:

Отсутствие обратных связей между этапами. Не соответствует реальным условиям разработки программного продукта

V-модель (V-model) - разработка через тестирование



V-модель – это улучшенная версия классической каскадной модели. Здесь на каждом этапе происходит контроль текущего процесса, для того чтобы убедиться в возможности перехода на следующий уровень. В этой модели тестирование начинается еще со стадии написания требований, причем для каждого последующего этапа предусмотрен свой уровень тестового покрытия.

Для каждого уровня тестирования разрабатывается отдельный тест-план, то есть во время тестирования текущего уровня мы также занимаемся разработкой стратегии тестирования следующего. Создавая тест-планы, мы также определяем ожидаемые результаты тестирования и указываем критерии входа и выхода для каждого этапа.

В V-модели каждому этапу проектирования и разработки системы соответствует отдельный уровень тестирования. Здесь процесс разработки представлен нисходящей последовательностью в левой части условной буквы V, а стадии тестирования – на ее правом ребре. Соответствие этапов разработки и тестирования показано горизонтальными линиями.

Плюсы и минусы V-модели:

- + строгая этапизация;
- + планирование тестирования и верификация системы производятся на ранних этапах;
- + улучшенный, по сравнению с каскадной моделью, тайм-менеджмент;
- + промежуточное тестирование.
- недостаточная гибкость модели;
- собственно создание программы происходит на этапе написания кода, то есть уже в середине процесса разработки;
- недостаточный анализ рисков;
- нет работы с параллельными событиями и возможности динамического внесения изменений.

Когда использовать V-модель:

- В проектах, в которых существуют временные и финансовые ограничения;
- Для задач, которые предполагают более широкое, по сравнению с каскадной моделью, тестовое покрытие.

Итеративная модель (Iterative model)



Не все модели жизненного цикла последовательны. Существуют также итеративные (или инкрементальные) модели, в которых используется другой подход. Вместо одной продолжительной последовательности действий здесь весь жизненный цикл продукта разбит на ряд отдельных мини-циклов. Причем каждый из них состоит из все тех же базовых стадий модели жизненного цикла. Эти мини-циклы называются итерациями. В каждой из итераций происходит разработка отдельного компонента системы, после чего этот компонент добавляется к уже ранее разработанному функционалу.

Итеративная модель не предполагает полного объема требований для начала работ над продуктом. Разработка программы может начинаться с требований к части функционала, которые могут впоследствии дополняться и изменяться. Процесс повторяется, обеспечивая создание новой версии продукта для каждого цикла.

В несколько упрощенном виде, итеративная модель состоит из четырех основных стадий, которые повторяются в каждой из итераций (**plan-do-check-act**):

- определение и анализ требований;
- дизайн и проектирование – согласно требованиям. Причем дизайн может как разрабатываться отдельно для данной функциональности, так и дополнять уже существующий;

- разработка и тестирование – кодирование, интеграция и тестирование нового компонента;
- фаза ревью – оценка, пересмотр текущих требований и предложения дополнений к ним.

По результатам каждой итерации принимается решение – будут ли использованы ее результаты для дополнения существующей функциональности в качестве входной точки для начала следующей итерации (т.н. инкрементальное прототипирование). В конечном итоге, достигается точка, в которой все требования были воплощены в продукте – происходит релиз.

Ключ к успешному использованию этой модели – строгая валидация требований и тщательная верификация разрабатываемой функциональности в каждой из итераций.

Основные стадии процесса разработки в итеративной модели фактически повторяют модель водопада. В каждой итерации создается программное обеспечение, требующее тестирования на всех уровнях.

Плюсы и минусы итеративной модели:

- + раннее создание работающего ПО;
- + гибкость – готовность к изменению требований на любом этапе разработки;
- + каждая итерация – маленький этап, для которого тестирование и анализ рисков обеспечить проще, чем для всего жизненного цикла продукта.
- каждая фаза – самостоятельна, отдельные итерации не накладываются;
- могут возникнуть проблемы с реализацией общей архитектуры системы, поскольку не все требования известны к началу проектирования.

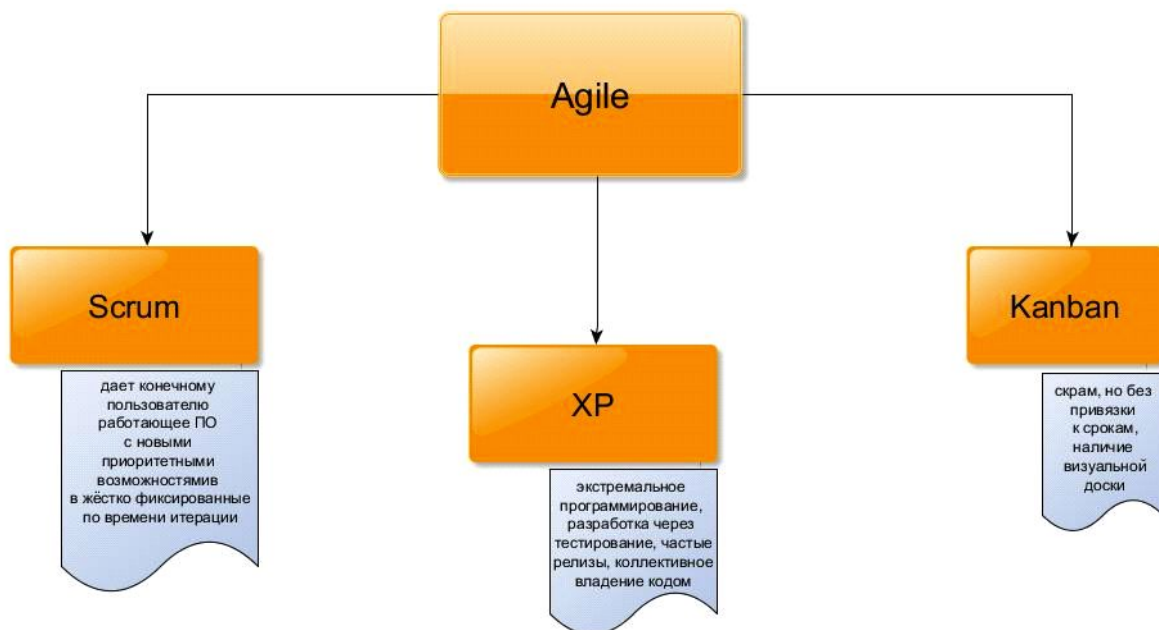
Когда использовать итеративную модель:

- для крупных проектов;
- когда известны, по крайней мере, ключевые требования;
- когда требования к проекту могут меняться в процессе разработки.

Вместе с гибкостью и возможностью быстро реагировать на изменения, итеративные модели приносят дополнительные сложности в управление проектом и отслеживание его хода. При использовании итеративного подхода значительно сложнее становится

адекватно оценить текущее состояние проекта и спланировать долгосрочное развитие событий, а также предсказать сроки и ресурсы, необходимые для обеспечения определенного качества результата.

Agile – семейство гибких методологий разработки.



Scrum - одна из самых популярных методологий гибкой разработки. Одна из причин ее популярности - простота. В Scrum всего три роли.

- Scrum Master
- Product Owner
- Team

Скрам Мастер (Scrum Master) - самая важная роль в методологии. Скрам Мастер отвечает за успех Scrum в проекте. По сути, Скрам Мастер является интерфейсом между менеджментом и командой. В Agile команда является самоорганизующейся и самоуправляемой.

Основные обязанности Скрам Мастера таковы:

- Создает атмосферу доверия
- Участвует в митингах

- Устраняет препятствия
- Делает проблемы и открытые вопросы видимыми
- Отвечает за соблюдение практик и процесса в команде

Скрам Мастер ведет Daily Scrum Meeting и отслеживает прогресс команды при помощи Sprint Backlog, отмечая статус всех задач в спринте.

Product Owner - это человек, отвечающий за разработку продукта. Как правило, это product manager для продуктовой разработки, менеджер проекта для внутренней разработки и представитель заказчика для заказной разработки. Product Owner - это единая точка принятия окончательных решений для команды в проекте.

Обязанности Product Owner таковы:

- Отвечает за формирование product vision
- Управляет ROI
- Управляет ожиданиями заказчиков и всех заинтересованных лиц
- Координирует и приоритизирует Product backlog
- Предоставляет понятные и тестируемые требования команде
- Взаимодействует с командой и заказчиком
- Отвечает за приемку кода в конце каждой итерации

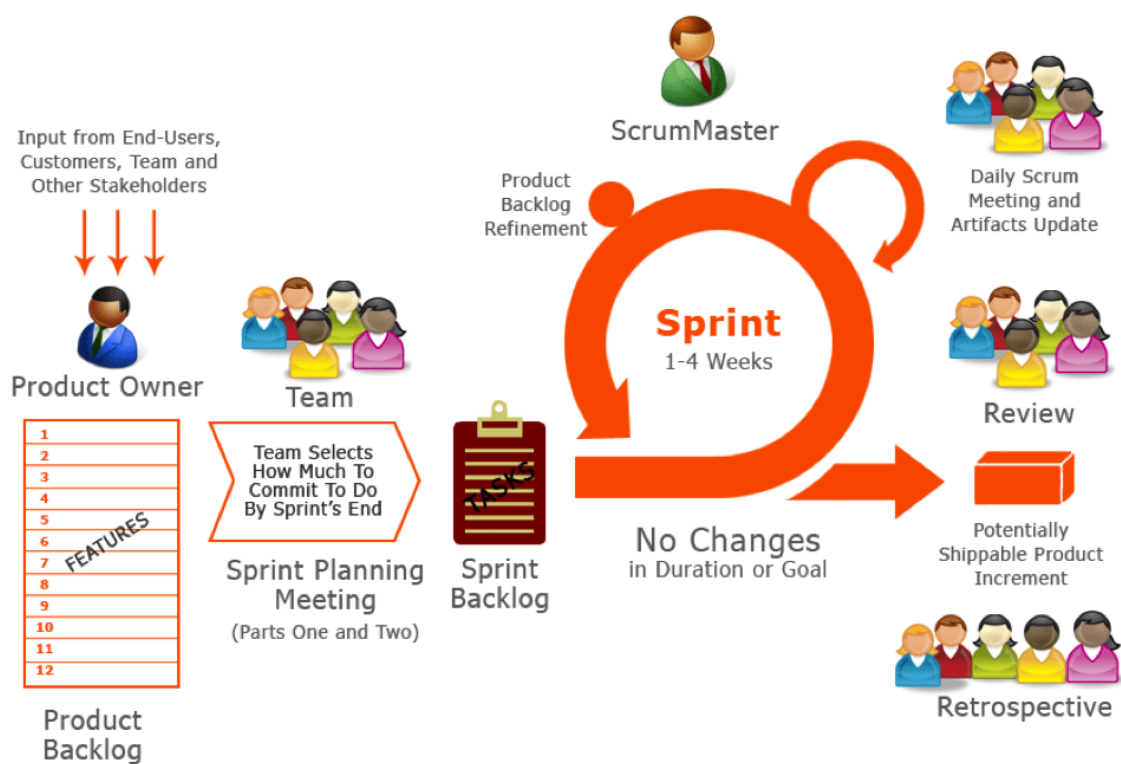
Product Owner ставит задачи команде, но он не вправе ставить задачи конкретному члену проектной команды в течении спринта.

Команда (Team). В методологии Scrum команда является самоорганизующейся и самоуправляемой. Команда берет на себя обязательства по выполнению объема работ на спринт перед Product Owner. Работа команды оценивается как работа единой группы. В Scrum вклад отдельных членов проектной команды не оценивается, так как это разваливает самоорганизацию команды.

Обязанности команды таковы:

- Отвечает за оценку элементов бэклога
- Принимает решение по дизайну и имплементации
- Разрабатывает софт и предоставляет его заказчику
- Отслеживает собственный прогресс (вместе со Скрам Мастером).
- Отвечает за результат перед Product Owner

Размер команды ограничивается размером группы людей, способных эффективно взаимодействовать лицом к лицу. Типичные размер команды - 7 плюс минус 2.



5.4. Артефакты

Product Backlog - это приоритезированный список имеющихся на данный момент бизнес-требований и технических требований к системе. Product Backlog постоянно пересматривается и дополняется - в него включаются новые требования, удаляются ненужные, пересматриваются приоритеты. За Product Backlog отвечает Product Owner. Он также работает совместно с командой для того, чтобы получить приближенную оценку на выполнение элементов Product Backlog для того, чтобы более точно расставлять приоритеты в соответствии с необходимым временем на выполнение.

Sprint Backlog содержит функциональность, выбранную Product Owner из Product Backlog. Все функции разбиты по задачам, каждая из которых оценивается командой. Каждый день команда оценивает объем работы, который нужно проделать для завершения задач.

В Scrum итерация называется Sprint. Ее длительность составляет 3-6 недель. Результатом Sprint является готовый продукт (build), который можно передавать (deliver) заказчику (по крайней мере, система должна быть готова к показу заказчику). Короткие спринты обеспечивают быстрый feedback проектной команде от заказчика. Заказчик получает возможность гибко управлять scope системы, оценивая результат спринта и предлагая улучшения к созданной функциональности. Такие улучшения попадают в Product Backlog, приоритизируются наравне с прочими требованиями и могут быть запланированы на следующий (или на один из следующих) спринтов. Каждый спринт представляет собой маленький "водопад". В течение спринта делаются все работы по сбору требований, дизайну, кодированию и тестированию продукта.

Планирование спринта (Sprint Planning Meeting) происходит в начале новой итерации Спринта. Из резерва проекта выбираются задачи, обязательства по выполнению которых за спринт принимает на себя команда. На основе выбранных задач создается резерв спринта. Каждая задача оценивается в поинтах. Обсуждается и определяется, каким образом будет реализован этот объем работ. Продолжительность совещания ограничена сверху 4-8 часами в зависимости от продолжительности итерации, опыта команды и т. п.

Ежедневное совещание (Daily Scrum meeting) начинается точно вовремя; длится не более 15 минут. В течение совещания каждый член команды отвечает на 3 вопроса:

- Что сделано с момента предыдущего ежедневного совещания?
- Что будет сделано с момента текущего совещания до следующего?
- Какие проблемы мешают достижению целей спринта?

Ретроспективное совещание (Retrospective meeting) проводится после завершения спринта. Члены команды высказывают своё мнение о прошедшем спринте.

Отвечают на два основных вопроса:

- Что было сделано хорошо в прошедшем спринте?
- Что надо улучшить в следующем?

Выполняют улучшение процесса разработки (решают вопросы и фиксируют удачные решения). Ограничена одним—тремя часами.

УРОК 6.

6.1. Сети. IP адрес и порт

Возникает вопрос - каким образом клиент и сервер “находят” друг друга в сети. И что, собственно говоря, представляет из себя сеть.



Общее определение гласит, что сеть - это система, обеспечивающая обмен данными между вычислительными устройствами (компьютеры, серверы, маршрутизаторы и другое оборудование). На сегодняшний день все мы являемся активными пользователями сети интернет.

Для того, чтобы устройства могли обмениваться информацией по сети существует стек протоколов, который регламентирует эти механизмы. Наиболее популярный стек протоколов это **TCP/IP** — набор сетевых протоколов передачи данных, используемых в сетях, включая сеть Интернет. Название TCP/IP происходит из двух наиважнейших протоколов семейства — Transmission Control Protocol (TCP) и Internet Protocol (IP), которые были разработаны и описаны первыми в данном стандарте.

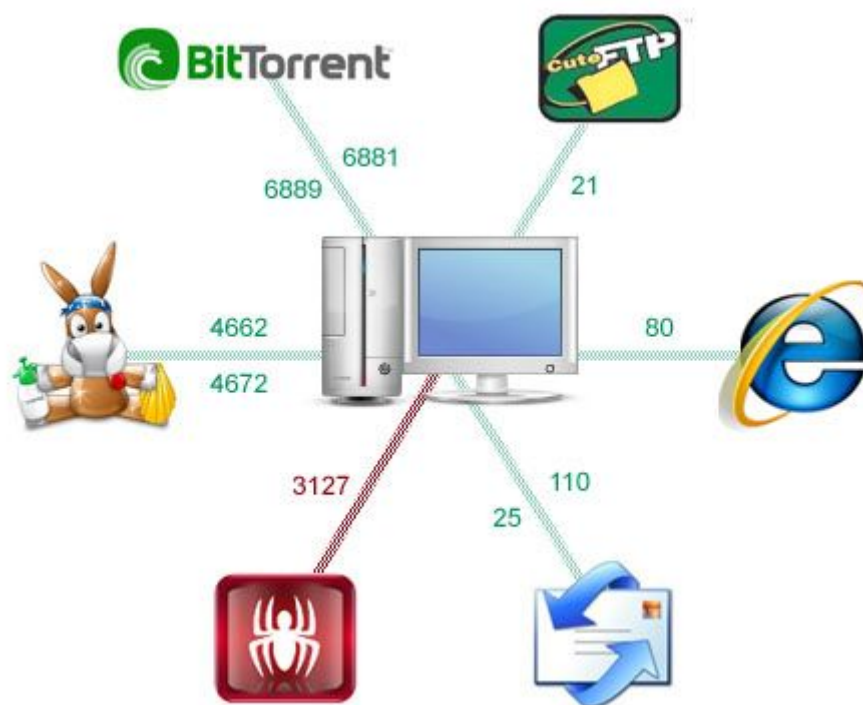
Подобно домашним адресам для переписки по почте, каждое сетевое устройство имеет свой адрес - адрес, который служит для идентификации этого устройства в сети и построения оптимального маршрута пакетов.

В сети Интернет это IP(Internet protocol) - уникальный сетевой адрес узла в компьютерной сети, построенной по протоколу IP. В сети Интернет требуется глобальная уникальность адреса; в случае работы в локальной сети требуется уникальность адреса в пределах сети. В версии протокола IPv4 IP-адрес имеет длину 4 байта, а в версии протокола IPv6 IP-адрес имеет длину 16 байт.

Удобной формой записи IP-адреса (IPv4) является запись в виде четырёх десятичных чисел значением от 0 до 255, разделенных точками, например, 192.168.0.3.

Таким образом, используя IP адрес, один компьютер может передавать информацию другому. А как удаленный компьютер может передать информацию конкретной программе на вашем компьютере? Программ, работающих на компьютере много, а IP-адрес один. Для адресации конкретной программе существует такое понятие как порт.

Порт - это условное натуральное число. Программа может "открыть" порт - это значит заявить во всеуслышание что "если придет информация на порт 5190 - знайте, это для меня". "Закрыть порт" - значит освободить его для другой программы и больше не получать информацию. Итак, сетевой порт — условное число от 1 до 65535, указывающее, какому приложению предназначается пакет.



Многие программы, которые работают с сетью, рассчитаны на подключение к определенным портам. Как видно из рисунка, интернет браузеры, например Internet Explorer, используют в своей работе порт 80. Это означает, что когда вы пишете, например, <http://ukr.net>, то компьютер ваш обращается к серверу ukr.net (ну то есть сначала узнает его IP-адрес), а потом сообщает, что будет ждать ответ на 80 порту. Почтовые программы, например Outlook Express, используют 2 порта (обычно, но не всегда), для отправки почты порт 25 и для приема 110 порт. Если установить программу для обмена файлами emule, то она откроет, необходимые для своей работы порты 4662 и 4672.

Таким образом, если компьютер в сети не защищен, то каждая программа, установленная на нем, сможет открыть любой необходимый ей порт. Точно также любая программа извне может подключиться к любому открытому порту компьютера. **Брандмауэр** (межсетевой экран), помимо всего прочего, следит, какие порты открываются, и выдает разрешение на открытие. Когда порт открыт - кто угодно может посылать на него информацию. В частности, злоумышленник может написать туда что-нибудь такое, от чего программа, слушающая этот порт, может быть взломана. Но, не открыв порт, программа не сможет работать по назначению.

Использование портов позволяет независимо использовать TCP протокол сразу многим приложениям на одном и том же компьютере. Практически всегда клиент начинает исходящие соединения, а сервер ожидает входящих соединений (от клиентов), хотя бывают и исключения. Сервер при запуске сообщает Операционной Системе, что хотел бы «занять» определенный порт (или несколько портов). После этого все пакеты, приходящие на компьютер к этому порту, ОС будет передавать этому серверу. Говорят, что сервер «слушает» этот порт. Клиент, начиная соединение, запрашивает у своей ОС какой-нибудь незанятый порт во временное пользование, и указывает его в посланных пакетах как порт источника. Затем на этот порт он получит ответные пакеты от сервера.

Таким образом,

Сервер:

- слушает на определённом порту, заранее известном клиенту
- занимает этот порт на всё время, пока не завершит работу
- об IP адресе и номере порта клиента узнаёт из приглашения, посланного клиентом

Клиент:

- заранее знает IP адрес и порт сервера
- выбирает у себя произвольный порт, который освобождает после окончания соединения - посылает приглашение к соединению

6.2. Архитектура клиент-сервер

«Клиент — сервер» (client-server) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами.

Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы — сервера, ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных(например загрузка файлов(посредством HTTP, FTP, BitTorrent), потоковое

мультимедиа или работа с базами данных) или в виде сервисных функций(например работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине).

Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, ее также называют сервером, а машины, выполняющие клиентские программы соответственно - клиентами.

Другими словами, компьютеры и программы, входящие в состав информационной системы, не являются равноправными. Некоторые из них владеют ресурсами (файловая система, процессор, принтер, база данных и т.д.), другие имеют возможность обращаться к этим ресурсам. Компьютер (или программу), управляющий ресурсом, называют сервером этого ресурса (файл- сервер, сервер базы данных, вычислительный сервер...). Клиент и сервер какого-либо ресурса могут находиться как на одном компьютере, так и на различных компьютерах, связанных сетью.

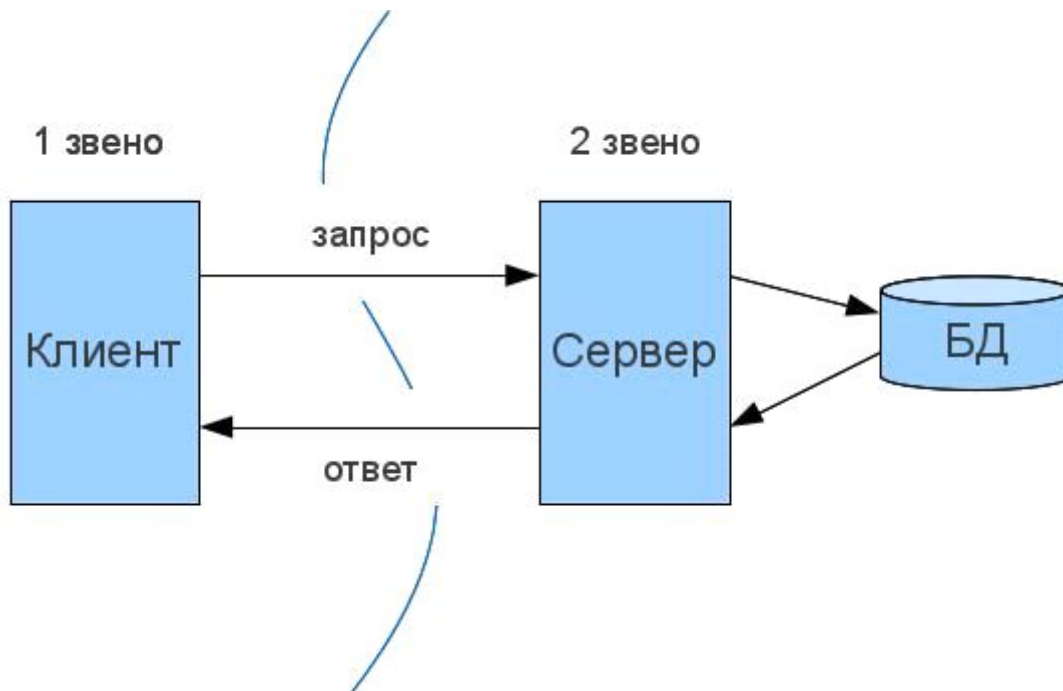
В рамках многоуровневого представления вычислительных систем можно выделить три группы функций, ориентированных на решение различных подзадач:

- функции ввода и отображения данных (обеспечивают взаимодействие с пользователем);
- прикладные функции, характерные для данной предметной области;
- функции управления ресурсами (файловой системой, базой данных и т.д.)

Архитектура «клиент-сервер» определяет общие принципы организации взаимодействия в сети, где имеются серверы, узлы-поставщики некоторых специфичных функций (сервисов) и клиенты, потребители этих функций.

Двухзвенная архитектура

В любой сети (даже одноранговой), построенной на современных сетевых технологиях, присутствуют элементы клиент-серверного взаимодействия, чаще всего на основе двухзвенной архитектуры. Двухзвенной (two-tier, 2-tier) она называется из-за необходимости распределения трех базовых компонентов между двумя узлами (клиентом и сервером).



Двухзвенная архитектура используется в клиент-серверных системах, где сервер отвечает на клиентские запросы напрямую и в полном объеме, при этом используя только собственные ресурсы. Т.е. сервер не вызывает сторонние сетевые приложения и не обращается к сторонним ресурсам для выполнения какой-либо части запроса.

Трёхзвенная архитектура

Еще одна тенденция в клиент-серверных технологиях связана со все большим использованием распределенных вычислений. Они реализуются на основе модели сервера приложений, где сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее

согласованном формате. В этом случае двухзвенная клиент-серверная архитектура становится трехзвенной (three-tier, 3-tier).

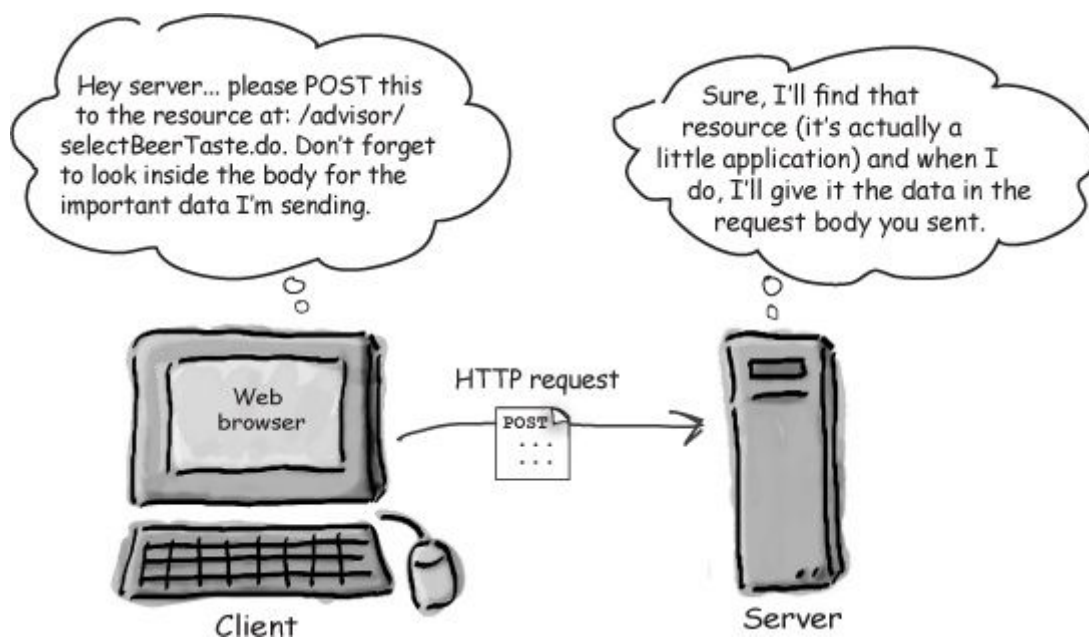


Трехзвенная архитектура сложнее, но благодаря тому, что функции распределены между серверами второго и третьего уровня, эта архитектура представляет: Высокую степень гибкости и масштабируемости. Высокую безопасность (т.к. защиту можно определить для каждого сервиса или уровня). Высокую производительность (т.к. задачи распределены между серверами).

Итак, основная идея архитектуры «клиент-сервер» состоит в разделении сетевого приложения на несколько компонентов, каждый из которых реализует специфический набор сервисов. Компоненты такого приложения могут выполняться на разных компьютерах, выполняя серверные и/или клиентские функции. Это позволяет повысить надежность, безопасность и производительность сетевых приложений и сети в целом.

А теперь на пальцах - когда мы используем клиент-сервер ? В любой момент работы с интернетом мы являемся клиентом и используем сервера и ресурсы тех сайтов, на которые ходим.

Допустим мы заходим на новостной сайт. В этот момент наш браузер формирует запрос к серверу сайта. Сервер формирует ответ в котором будет содержаться вся необходимая информация - вид веб страницы, текст, картинки, видео. Наш браузер получает эту информацию и отображает ее нам.



Таким образом новости хранятся всего лишь в одном месте, но люди со всего мира имеют к ним доступ.

Теперь мы решили написать комментарий к новости. В момент нажатия "Отправить" - наш комментарий будет отправлен на сервер новостного ресурса и сохранен там. И теперь любой другой пользователь сети сможет увидеть наш комментарий.

Более подробно и с техническими деталями тут :

<https://habrahabr.ru/company/htmlacademy/blog/254825/>

6.3. Консоль (интерфейс командной строки)

Консоль компьютера (англ. *console* — пульт управления) — совокупность устройств (в том числе устройств ввода-вывода), обеспечивающая взаимодействие человека-оператора с компьютером.

Современные консоли

В большинстве современных компьютеров консолью является комплект устройств интерактивного ввода-вывода, присоединенных к компьютеру непосредственно (не через сеть): дисплей, клавиатура, мышь. Консольный сеанс в многопользовательских операционных системах — это сеанс, осуществляемый человеком, сидящим непосредственно перед компьютером (в противоположность сеансу удаленного доступа, например через telnet, ssh, X Window System, RDP и т. п.). Данная трактовка термина *консоль* безотносительна к типу пользовательского интерфейса: текстовому (CUI) или графическому (GUI).

Программное обеспечение

- В различных программах и играх консолью стали называть окно для вывода системных сообщений и приема команд (интерфейс командной строки).
- Консолью называют программное обеспечение, реализующее текстовый интерфейс.

Мы рассмотрим именно программную консоль - интерфейс командной строки, так как это необходимый инструмент для тестировщика.

Интерфейс командной строки (англ. *Command line interface*, **CLI**) — разновидность текстового интерфейса (**TUI**) между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (*команд*), в UNIX-системах возможно применение мыши. Также известен под названием *консоль*.

Интерфейс командной строки противопоставляется системам управления программой на основе меню, а также различным реализациям графического интерфейса.

Формат вывода информации в интерфейсе командной строки не регламентируется; обычно это простой текстовый вывод, но может быть и графическим, звуковым и т. д.

Интерфейс командной строки применяется по таким причинам:

- Небольшой расход памяти по сравнению с системой меню.

- В современном программном обеспечении имеется большое число команд, многие из которых нужны крайне редко. Поэтому даже в некоторых программах с графическим интерфейсом применяется командная строка: набор команды (при условии, что пользователь знает эту команду) осуществляется гораздо быстрее, чем, например, навигация по меню.
- Естественное расширение интерфейса командной строки — пакетный интерфейс. Его суть в том, что в файл обычного текстового формата записывается последовательность команд, после чего этот файл можно выполнить в программе, что будет равносильно выполнению этих команд руками. Примеры — .bat-файлы в DOS и Windows, shell-скрипты в Unix-системах. **Это часто используется в автоматизированном тестировании и системах непрерывной интеграции.**

Если программа полностью или почти полностью может управляться командами интерфейса командной строки и поддерживает пакетный интерфейс, умелое сочетание интерфейса командной строки с графическим предоставляет пользователю очень мощные возможности.

Формат команды

Наиболее общий формат команд (в квадратные скобки помещены необязательные части):

[символ_начала_команды]имя_команды [параметр_1 [параметр_2 [...]]]

Символ начала команды может быть самым разным, однако чаще всего для этой цели используется косая черта, которая в математических операциях означает деление (/). Если строка вводится без этого символа, выполняется некоторая базовая команда: например, строка «Привет» в [IRC](#) эквивалентна вводу «/msg Привет». Если же такой базовой команды нет, символ начала команды отсутствует вообще (как, например, в [DOS](#)).

Параметры команд могут иметь самый разный формат. В основном применяются следующие правила:

- параметры разделяются пробелами (и отделяются от названия команды пробелом)
- параметры, содержащие пробелы, обрамляются кавычками-апострофами (') или двойными кавычками (")
- если параметр используется для обозначения включения какой-либо опции, выключенной по умолчанию, он начинается с косой черты (/) или дефиса (-)

- если параметр используется для включения/выключения какой-либо опции, он начинается (или заканчивается) знаком плюс или минус (для включения и выключения соответственно)
- если параметр указывает действие из группы действий, назначенных команде, он не начинается со специальных символов
- если параметр указывает объект, к которому применяется действие команды, он не начинается со специальных символов
- если параметр указывает дополнительный параметр какой-либо опции, то он имеет формат /опция: дополнительный_параметр (вместо косой черты также может употребляться дефис).

Применение командной строки:

- Интерфейс операционной системы
- Компьютерные игры

Изначально консоль в играх использовалась для отладки.

Как только появился интерфейс командной строки, стали появляться и игры, его использующие. Особенно актуально это было на тех платформах, где более сложные интерфейсы (графические) было невозможно реализовать вследствие аппаратных ограничений.

- Другие программы:
 - САПР (AutoCAD)
 - текстовые редакторы (Vim)
 - Браузеры (Vimperator — расширение для браузера Firefox, позволяющее управлять им, как редактором Vim)

Достоинства:

- Легкость автоматизации. Shell script в UNIX-подобных системах является полноценным интерпретируемым языком программирования и способен автоматизировать любую системную задачу. В Windows присутствует их примитивный аналог — пакетные файлы, и более мощный аналог — powershell. С графическим интерфейсом без поддержки программой командной строки это сделать почти невозможно.
- Можно управлять программами, не имеющими графического интерфейса (например, выделенным сервером).

- Любую команду можно вызвать небольшим количеством нажатий.
- Можно обращаться к командам для разных исполняемых файлов почти мгновенно и непосредственно, тогда как в GUI приходится сначала запускать, а затем закрывать графический интерфейс для каждого исполняемого файла.
- Просмотрев содержимое консоли, можно повторно увидеть промелькнувшее сообщение, которое вы не успели прочитать.
- Можно пользоваться удаленным компьютером с любого устройства подключенного к Интернету или локальной сети без особых затрат трафика и потерь в быстродействии.

Недостатки:

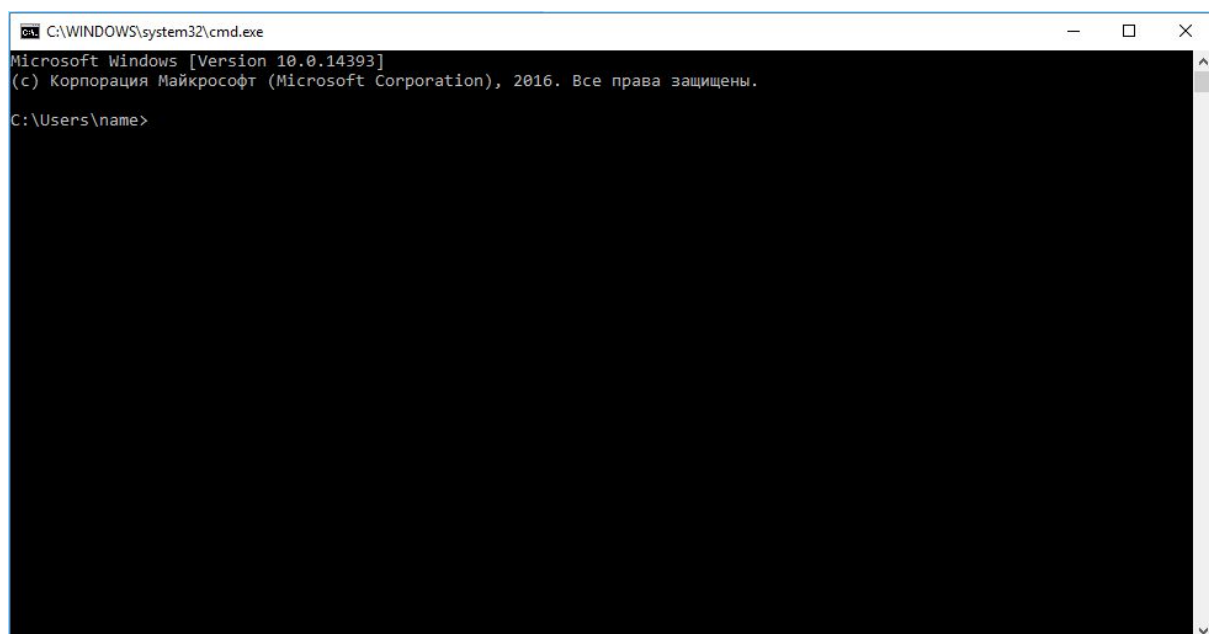
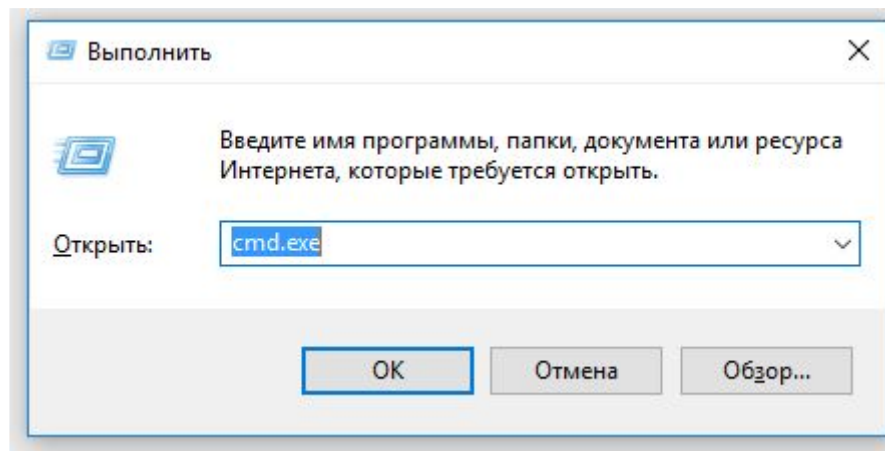
- Интерфейс командной строки не является «дружелюбным» для пользователей, которые начали знакомство с компьютером с графического режима
- Необходимость изучения синтаксиса команд и запоминания сокращений
- Без автодополнения ввод длинных и содержащих спецсимволы параметров с клавиатуры может быть затруднительным.
- Отсутствие «аналогового» ввода. Например подбор громкости с помощью озвученного ползунка позволяет выставить подходящую громкость быстрее, чем командой вроде `amix -v 90`. (Однако, озвученный ползунок вполне может быть псевдографическим, что реализовано в большинстве консольных плееров).

Рассмотрим для примера несколько команд ОС Windows.

Первый вопрос который возникает: как открыть командную строку? В Windows это программа `cmd.exe`, всегда установленная по умолчанию.

Запускаем Выполнить (Run): Windows+R

Находим командную строку: `cmd.exe`



Также можно: Пуск → Все программы → Стандартные → Командная строка (Command Prompt).

После запуска утилиты можно получить справочную информацию о командах и формате их написания в консоли. Для этого нужно ввести оператор *help* и нажать на «*Enter*»:

```
C:\WINDOWS\system32\cmd.exe
C:\Users\name>
C:\Users\name>help
Для получения сведений об определенной команде наберите HELP <имя команды>
ASSOC      Вывод либо изменение сопоставлений по расширениям имен файлов.
ATTRIB     Отображение и изменение атрибутов файлов.
BREAK      Включение и выключение режима обработки комбинации клавиш CTRL+C.
BCDEDIT     Задаёт свойства в базе данных загрузки для управления начальной
            загрузкой.
CACLS      Отображение и редактирование списков управления доступом (ACL)
            к файлам.
CALL       Вызов одного пакетного файла из другого.
CD         Вывод имени либо смена текущей папки.
CHCP       Вывод либо установка активной кодовой страницы.
CHDIR      Вывод имени либо смена текущей папки.
CHKDSK     Проверка диска и вывод статистики.
CHKNTFS    Отображение или изменение выполнения проверки диска во время
            загрузки.
CLS        Очистка экрана.
CMD        Запуск еще одного интерпретатора командных строк Windows.
COLOR      Установка цветов переднего плана и фона, используемых по умолчанию.
COMP       Сравнение содержимого двух файлов или двух наборов файлов.
COMPACT    Отображение и изменение сжатия файлов в разделах NTFS.
CONVERT    Преобразует тома FAT в NTFS. Вы не можете
            преобразовать текущий диск.
COPY       Копирование одного или нескольких файлов в другое место.
DATE       Вывод либо установка текущей даты.
DEL        Удаление одного или нескольких файлов.
DIR        Вывод списка файлов и подпапок из указанной папки.
DISKPART   Отображает или настраивает свойства раздела диска.
DOSKEY     Редактирует командные строки, повторно вызывает команды Windows и создает
            макросы.
DRIVERQUERY Отображает текущее состояние и свойства драйвера устройства.
ECHO       Отображает сообщения и переключает режим отображения команд на экране.
ENDLOCAL   Завершает локализацию изменений среды для пакетного файла.
ERASE      Удаляет один или несколько файлов.
EXIT       Завершает работу программы CMD.EXE (интерпретатора командных строк).
FC         Сравнивает два файла или два набора файлов и
            отображает различия между ними.
FIND       Ищет текстовую строку в одном или нескольких файлах.
FINDSTR    Ищет строки в файлах.
FOR        Запускает указанную команду для каждого из файлов в наборе.
FORMAT     Форматирует диск для работы с Windows.
FSUTIL     Отображает или настраивает свойства файловой системы.
FTYPE      Отображает либо изменяет типы файлов, используемые при
            сопоставлении по расширениям имен файлов.
GOTO       Направляет интерпретатор команд Windows в отмеченную строку
            пакетной программы.
GPRESULT   Отображает информацию о групповой политике для компьютера или пользователя.
GRAFTABL   Позволяет Windows отображать расширенный набор символов в
            графическом режиме.
HELP       Выводит справочную информацию о командах Windows.
ICACLS     Отображает, изменяет, архивирует или восстанавливает
            списки ACL для файлов и каталогов.
IF         Выполняет условную обработку в пакетных программах.
LABEL      Создает, изменяет или удаляет метки тома для дисков.
MD         Создает каталог.
MKDIR      Создает каталог.
MKLINK     Создает символичные ссылки и жесткие связи
MODE       Настраивает системные устройства.
MORE       Последовательно отображает данные по частям размером в один экран.
MOVE       Перемещает один или несколько файлов из одного каталога
```

Также выполнение любой команды с параметром `--help` выводит на экран информацию о команде, например `shutdown --help`:

```
C:\WINDOWS\system32\cmd.exe
C:\Users\name>
C:\Users\name>shutdown --help
Использование: shutdown [/i | /l | /s | /r | /g | /a | /p | /h | /e | /o] [/hybrid] [/soft] [/fw] [/f]
[/m \\компьютер] [/t xxx] [/d [p|u]xx:yy [/c "комментарий"]]

Без пар. Отображение справки. То же, что и с параметром /?.
/? Отображение справки. То же, что и без параметров.
/i Отображение графического интерфейса пользователя.
Этот параметр должен быть первым.
/l Завершение сеанса. Этот параметр нельзя использовать с
параметрами /m или /d.
/s Завершение работы компьютера.
/r Полное завершение работы и перезагрузка компьютера.
/g Полное завершение работы и перезагрузка компьютера. Запуск всех
зарегистрированных приложений после перезагрузки системы.
/a Отмена завершения работы системы.
Этот параметр можно использовать только в период ожидания.
Объединить с /fw для сброса всех ожидающих загрузок во встроенное ПО.
/p Выключение локального компьютера без задержки или предупреждения.
Можно использовать с параметрами /d и /f.
/h Перевод локального компьютера в режим гибернации.
Можно использовать с параметром /f.
/hybrid Выполняет завершение работы компьютера и подготавливает его к быстрому запуску.
Необходимо использовать с параметром /s.
/fw Объединить с вариантом завершения работы, чтобы следующая загрузка перешла в
пользовательский интерфейс встроенного ПО.
/e Указание причины непредвиденного завершения работы компьютера.
/o Переход в меню дополнительных параметров загрузки и перезагрузка компьютера.
Необходимо использовать с параметром /r.
/m \\компьютер Указание конечного компьютера.
/t xxx Указание времени ожидания в xxx секунд до завершения работы компьютера.
Допустимый диапазон: 0-315360000 (10 лет); значение по умолчанию: 30.
Если задержка больше 0, подразумевается использование
параметра /f.
/c "комментарий" Комментарий с причиной перезагрузки или завершения работы.
Длина не должна превышать 512 знаков.
/f Принудительное закрытие запущенных приложений без предупреждения пользователей.
Подразумевается использование параметра /f, если
для параметра /t задано значение больше 0.
/d [p|u:]xx:yy Причина перезагрузки или завершения работы.
p означает запланированную перезагрузку или завершение работы.
u означает, что причина определяется пользователем.
Если не задано ни "p", ни "u", перезагрузка и завершение работы
не планируются.
xx - номер основной причины (целое положительное число меньше 256).
yy - номер дополнительной причины (целое положительное число меньше 65536).

Причины на этом компьютере:
(E = ожидалось, U = не ожидалось, P = планировалось, C = определено)
```

Для просмотра содержимого директории команда *dir*.


```
C:\WINDOWS\system32\cmd.exe
C:\Users\name>
C:\Users\name>
C:\Users\name>dir
Том в устройстве C не имеет метки.
Серийный номер тома: F471-D54B

Содержимое папки C:\Users\name

30.04.2017  11:48    <DIR>        .
30.04.2017  11:48    <DIR>        ..
30.03.2017  23:06    <DIR>        ansel
13.04.2017  02:12    <DIR>        Contacts
24.04.2017  21:12    <DIR>        Desktop
20.04.2017  00:22    <DIR>        Documents
23.04.2017  20:18    <DIR>        Downloads
13.04.2017  02:12    <DIR>        Favorites
29.03.2017  10:45    <DIR>        Intel
13.04.2017  02:12    <DIR>        Links
13.04.2017  02:12    <DIR>        Music
18.04.2017  21:30    <DIR>        OneDrive
13.04.2017  02:12    <DIR>        Pictures
13.04.2017  02:12    <DIR>        Saved Games
13.04.2017  02:12    <DIR>        Searches
29.03.2017  13:50    <DIR>        Tracing
30.04.2017  11:21    <DIR>        Videos
                0 файлов                0 байт
                17 папок  232 058 490 880 байт свободно

C:\Users\name>
```

Пример команды с параметрами:

shutdown -r -f -t 30

-r Полное завершение работы и перезагрузка компьютера.

-f Указание времени ожидания в xxx секунд до завершения работы компьютера.

-t Принудительное закрытие запущенных приложений без предупреждения пользователей.

То есть через 30 секунд компьютер закроет все запущенные приложения и перезагрузится.

Другие полезные команды, которые стоит знать:

cd - Вывод имени либо смена текущей папки

mkdir - Создает каталог

path - Отображает или устанавливает путь поиска исполняемых файлов.

find - Ищет текстовую строку в одном или нескольких файлах.

systeminfo - Отображает сведения о свойствах и конфигурации определенного компьютера.

Кроме стандартных команд, через командную строку удобно работать с различными утилитами, такими как:

ipconfig - Утилита командной строки для вывода деталей текущего соединения и управления клиентскими сервисами DHCP и DNS.

telnet - Утилита, реализующая клиентскую часть протокола telnet

netstat - Утилита, выводящая на дисплей состояние TCP-соединений (как входящих, так и исходящих), таблицы маршрутизации, число сетевых интерфейсов и сетевую статистику по протоколам.

С ними мы познакомимся подробнее на практическом занятии.

Описание команд доступных для ОС Windows:

<https://technet.microsoft.com/en-us/library/bb490890.aspx>

Самостоятельно рекомендуется изучить основные команды UNIX-подобных систем. Для этого желательно установить на виртуальной машине одну из UNIX-подобных ОС, например Ubuntu. Более простой способ - использовать эмулятор, чтобы потренироваться выполнять команды.

Эмулятор командной строки: <https://bellard.org/jslinux/>

Описание команд:

<https://www.linux.com/learn/how-use-linux-command-line-basics-cli>

http://help.ubuntu.ru/wiki/%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%BD%D0%B0%D1%8F_%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0

УРОК 7.

7.1. HTTP протокол

Мы уже упоминали о протоколах когда рассматривали тему клиент-сервер, теперь остановимся подробнее, в особенности рассмотрим HTTP протокол, так как это необходимо знать при тестировании веб-продуктов.

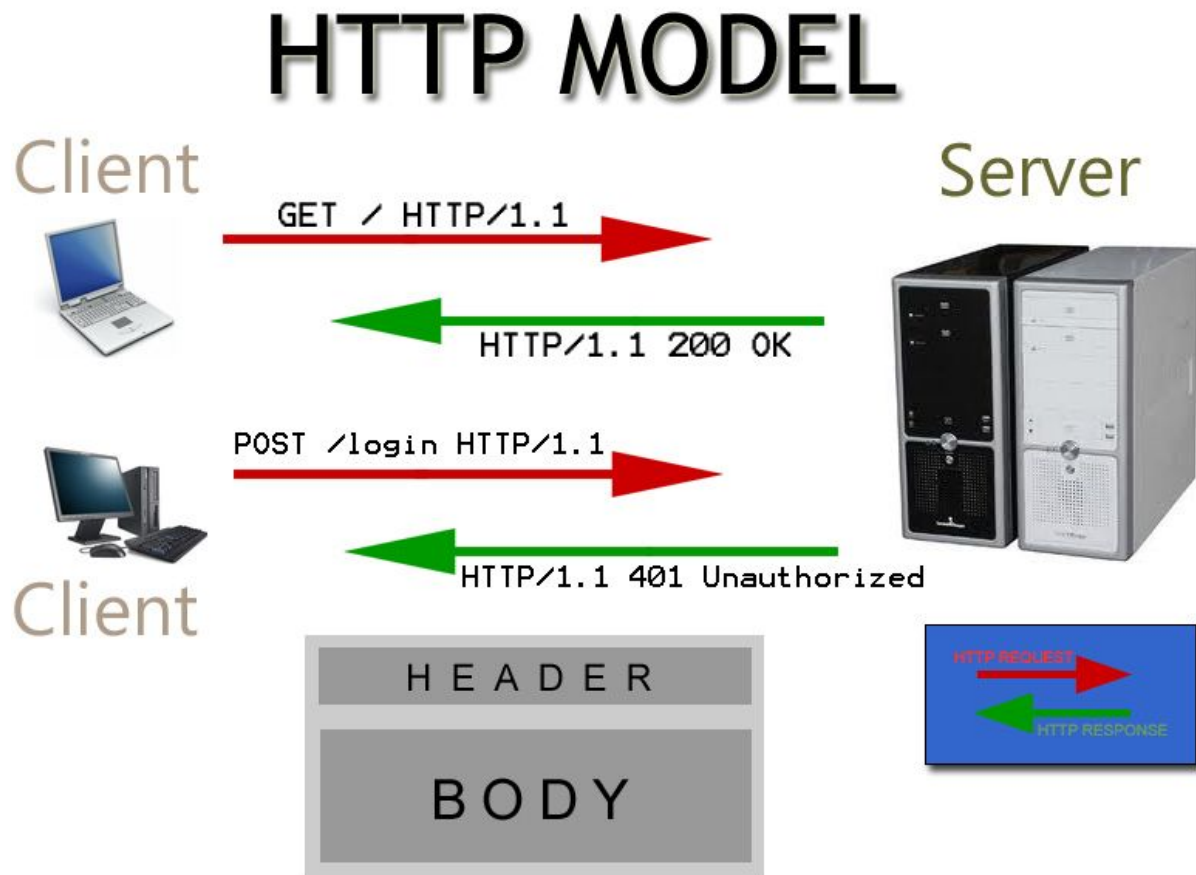
Немного истории:

Интернет получил свое начало 2 сентября 1969 года, когда ученым университета калифорнии впервые удалось передать данные между двумя компьютерами. Поначалу сеть носила имя ARPANET(Advanced Research Projects Agency Network). Целью было создать сеть связи, которая работала бы даже тогда, когда какая либо ее часть будет выведена из строя. Имелась в виду конечно угроза атомной войны. Первым создали соединение между университетом Калифорнии (Los Angeles) и Стэнфордским исследовательским институтом, а в конце 1969 года подключили к сети и Университет Калифорнии в Санта-Барбаре, и Университет Юты. К 1971 году были подключены уже 23 компьютера 15 различных учреждений, и к 1983 году компьютеров, подключенных к сети, было уже 500.

В 1973 году был придуман **FTP** (File Transfer Protocol) протокол, который позволял передачу данных между различными интернет-окружениями и позволял загрузку и публикацию веб-страниц на веб-сервере. 1989 году был придуман **гипертекст** (hypertext, текст, в котором путем какого-либо ключевого слова можно было перейти к связанному с ним документу). Таким образом начали объединять научные документы при помощи гипертекста - ссылаться. Надо четко понимать, что веб-страницы хранятся на веб-сервере, который передает их остальному миру.

Страницы размещаются на сервере в специальных каталогах, к которым есть доступ на чтение и у пользователей, обращающихся снаружи. Для передачи веб-страниц клиентам (веб-браузеру) веб-сервер использует протокол **HTTP** (HyperText Transfer Protocol). У каждой веб-страницы и другого ресурса в веб-сети есть свой адрес URL (Uniform Resource Locator), который состоит из доменного имени и, при необходимости, из подкаталогов и/или имени файла, а также из обозначения используемого протокола, который прописывается в начале этой строки (в случае веб-страниц http/https).

Когда вы заходите в браузер, не важно, какой именно браузер у вас установлен, и вводите в адресную строку адрес к сайту, браузер автоматически прибавляет к адресу приставку «http://». Единственное, эта приставка может быть по умолчанию скрыта, но если скопировать адрес и вставить его в другое место, то ее без труда можно будет увидеть. Эта приставка означает, что вы будете обращаться к ресурсу по протоколу HTTP. Основная задача протокола HTTP – это прием и передача гипертекстовых документов. Т.е. тех веб-страниц, которые мы просматриваем в браузере. Сторона, которая принимает содержимое веб-страниц — браузер (еще его называют клиентом), а сторона, которая отдает содержимое веб-страниц сервер. Технологию, по которой происходит этот обмен, называют «клиент- серверной» технологией. По сути, протокол HTTP – это инструмент, с помощью которого можно передавать веб-страницы в сети Интернет. А, что собой представляет веб- страница, которую мы получаем в ответе сервера? На самом деле это обычный HTML-код, который получает браузер и соответствующим образом его интерпретирует.



Работает это следующим образом: клиент (браузер) посылает запросы на сервер (apache, nginx, lighttpd) и в случае успешного соединения, в ответ на запрос сервер отправляет клиенту информацию. Под «информацией», в случае с HTTP, можно понимать практически всё что угодно. Связано это с упомянутой «гибкостью»: протокол способен передавать потоковый звук и видео, или же может выступать в виде транспорта для других протоколов. Теперь рассмотрим, как именно происходит коммуникация между клиентом и сервером. Мы уже определили, что клиент посылает запрос, который состоит из 3-х частей:

- Стартовая строка — в ней указывается тип сообщения.
- Заголовки — описывают тело сообщения, определяя его параметры.
- Тело сообщения — само сообщение, должно отделяться пустой строкой.

Методы HTTP

Методы в HTTP это своего рода инструкции, передаваемые в сообщении серверу. Синтаксически — это специально зарезервированное слово на английском языке. Особое внимание нужно обратить на регистр: метод пишется всегда в верхнем регистре. Кратко рассмотрим методы протокола HTTP:

1. **GET** - запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные. Выполнение одного и того же запроса приводит к одинаковому ответу.
2. **POST** - используется для отправки сущностей к определенному ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервер. Т.е. метод POST занимается отправкой данных, находящихся в теле сообщения, на сервер. Также часто используется для отправки информации из веб-форм и файлов. Простейший пример — написание поста на форуме или комментария в соцсети. После того, как вы написали ваше сообщение, браузер формирует POST-запрос и в его тело помещает ваше сообщение, а затем отправляет его на сервер. Выполнение одного и того же запроса может приводить к разным ответам.
3. **HEAD** - запрашивает ресурс так же, как и метод GET, но без тела ответа. Он запрашивает у сервера только мета-данные, и, соответственно, из ответа исключается тело сообщения. Пример использования - получение информации о файлах без скачивания. Также методом HEAD можно проверить валидность URI.
4. **PUT** - используется для загрузки данных на указанный URL. Данный метод предпочтительно использовать при передаче больших объемов информации.
5. **DELETE** - удаляет указанный ресурс.
6. **CONNECT** - устанавливает "туннель" к серверу, определенному по ресурсу.
7. **OPTIONS** - используется для описания параметров соединения с ресурсом.
8. **TRACE** - выполняет вызов возвращаемого тестового сообщения с ресурса.
9. **PATCH** - используется для частичного изменения ресурса.

Протокол может быть как http для обычных соединений, так и https для более безопасного обмена данными. Порт по умолчанию - 80. Далее следует путь к ресурсу на сервере и цепочка параметров.

Код Состояния

В ответ на запрос от клиента, сервер отправляет ответ, который содержит, в том числе, и код состояния. Данный код несет в себе особый смысл для того, чтобы клиент мог точнее понять, как интерпретировать ответ.

Код состояния информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Набор кодов состояния является стандартом, и все они описаны в соответствующих документах RFC.

Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие - порядковый номер состояния. За кодом ответа обычно следует краткое описание на английском языке.



Клиент может не знать все коды состояния, но он обязан отреагировать в соответствии с классом кода.

1xx

Информационные сообщения

Набор этих кодов был введен в HTTP/1.1. Сервер может отправить запрос вида: Expect: 100-continue, что означает, что клиент еще отправляет оставшуюся часть запроса. Клиенты, работающие с HTTP/1.0 игнорируют данные заголовки.

2xx

Сообщения об успехе.

Если клиент получил код из серии 2xx, то запрос ушел успешно. Самый распространённый вариант - это 200 OK. При GET запросе, сервер отправляет ответ в теле сообщения. Также существуют и другие возможные ответы:

202 Accepted: запрос принят, но может не содержать ресурс в ответе. Это полезно для асинхронных запросов на стороне сервера. Сервер определяет, отправить ресурс или нет.

204 No Content: в теле ответа нет сообщения.

205 Reset Content: указание серверу о сбросе представления документа.

206 Partial Content: ответ содержит только часть контента. В дополнительных заголовках определяется общая длина контента и другая информация.

3xx

Перенаправление.

Своеобразное сообщение клиенту о необходимости совершить ещё одно действие. Самый распространённый вариант применения: перенаправить клиент на другой адрес.

301 Moved Permanently: ресурс теперь можно найти по другому URL адресу.

302 Moved Temporarily: Запрашиваемый ресурс временно находится по новому URL.

303 See Other: ресурс временно можно найти по другому URL адресу. Заголовок Location содержит временный URL.

304 Not Modified: сервер определяет, что ресурс не был изменён и клиенту нужно задействовать закешированную версию ответа. Для проверки идентичности информации используется ETag (хэш Сущности - Entity Tag).

4xx

Клиентские ошибки.

Данный класс сообщений используется сервером, если он решил, что запрос был отправлен с ошибкой.

404 Not Found: самый распространенный ответ. Это означает, что ресурс не найден на сервере.

400 Bad Request: запрос был сформирован неверно.

401 Unauthorized: для совершения запроса нужна аутентификация. Информация передается через заголовок Authorization.

403 Forbidden: сервер не открыл доступ к ресурсу.

409 Conflict: сервер не может до конца обработать запрос, т.к. пытается изменить более новую версию ресурса. Это часто происходит при PUT запросах.

5xx

Ошибки сервера

Ряд кодов, которые используются для определения ошибки сервера при обработке запроса.

500 Internal Server Error: самый распространенный. Внутренняя ошибка сервера.

501 Not Implemented: сервер не поддерживает запрашиваемую функциональность.

503 Service Unavailable: это может случиться, если на сервере произошла ошибка или он перегружен. Обычно в этом случае, сервер не отвечает, а время, данное на ответ, истекает.

Заголовки

Перейдем к рассмотрению следующего важного момента — заголовки. Они используются для передачи служебной информации.

Заголовки делятся на 4 группы:

- Основные заголовки — обязательно включаются в любое сообщение клиента и сервера.
- Заголовки запроса — можно встретить только в запросах от клиента.
- Заголовки ответа — можно встретить только в ответах от сервера.
- Заголовки сущности — описывают сущность каждого сообщения (может относиться как к клиенту, так и к серверу).

С помощью заголовков можно передавать или получать обширный спектр информации и параметров, которые могут оказаться полезными в решении многих задач. Все заголовки подробно описаны в спецификации RFC(ссылка в конце раздела). Однако можно вводить свои, только нужно следить за тем, чтобы не возникало дублирования, иначе возможны ошибки. Вот основные (общие) заголовки, которые встречаются и в запросах и в ответах:

- **Cache-Control** — параметры управления кэшированием.
- **Connection** — информация о соединении.
- **Date** — дата создания сообщения.
- **Pragma** — специфические опции для выполнения.
- **Transfer-Encoding** — перечень кодировок, примененных для формирования сообщения.
- **Upgrade** — перечень протоколов, с которыми может работать клиент. Сервер указывает один.
- **Via** — история прохождения запроса через прокси сервера, с указанием версии протокола.

Рассмотрим пример:

Запрос:

GET /index.php HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9b5) Gecko/2008050509 Firefox/3.0b5
Accept: text/html
Connection: close

Первая строка — это строка запроса, остальные — заголовки; тело сообщения отсутствует

Ответ:

HTTP/1.0 200 OK
Server: nginx/0.6.31
Content-Language: ru
Content-Type: text/html; charset=utf-8
Content-Length: 1234
Connection: close

... САМА HTML-СТРАНИЦА ...

Как и любой другой протокол, HTTP строго регламентируется спецификацией. Ознакомиться с ней можно здесь: <https://tools.ietf.org/html/rfc2616>

7.2. Инструменты разработчиков

Существует множество инструментов для мониторинга HTTP трафика, например Chrome Developers Tools, Firebug, Tamper Data. Рассмотрим вкратце, какие возможности они предоставляют на примере Mozilla Firebug и Tamper Data.

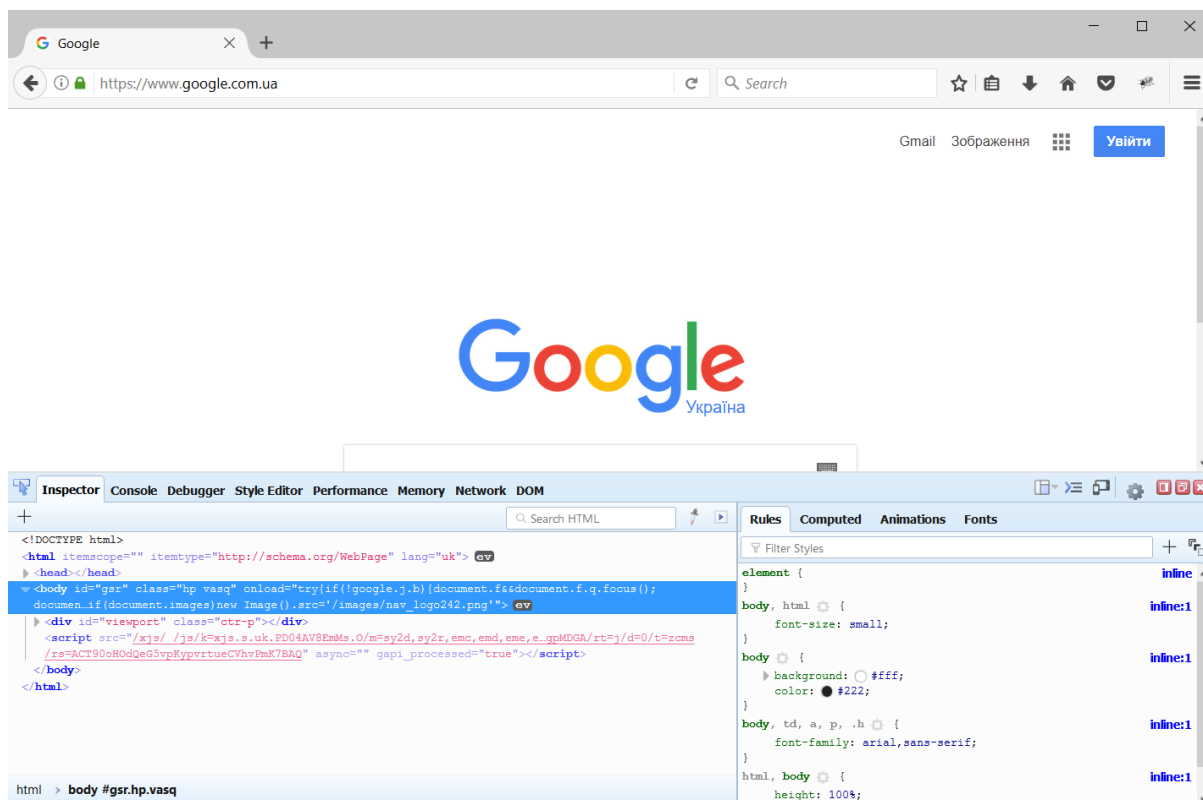
Firebug

Один клик мыши и перед вами откроется полный набор мощнейших инструментов Firebug. Firebug является расширением (Add-on) для Firefox.

Firebug предоставляет массу возможностей для веб-разработчика:

- возможность найти информацию о любом элементе;
- редактировать HTML-код и просматривать результаты редактирования в браузере;
- копировать различные аспекты элемента в буфер обмена, включая HTML-фрагмент;
- инспектировать любой CSS файл;
- показывать правила CSS, которые работают в каскаде, определяя стиль каждого элемента
- редактировать CSS и просматривать результаты редактирования в браузере;
- содержит полный словарь ключевых слов CSS в памяти;
- включает в себя мощный javascript отладчик;
- имеет командную строку для javascript, предоставляет набор мощных функций логирования;
- дает детализированную полезную информацию об ошибках в Javascript, CSS и XML;
- позволяет быстро находить нужные DOM-объекты и редактировать их.

Для открытия инструментов разработчика как Mozilla Firefox так и в Chrome достаточно комбинации клавиш Ctrl+Shift+I. Также можно открыть через выпадающее меню браузера.



Tamper Data

Tamper Data это также расширение браузера Mozilla Firefox для просмотра и изменения HTTP/HTTPS-заголовков. Tamper Data — Очень полезный плагин, который пригодится, как разработчикам, так и тестировщикам. Он позволяет вам анализировать все HTTP запросы которые протекают между вами и сервером. Но самое главное, вы можете легко и просто редактировать любой посланный заголовок. Этот хороший способ получить всю информацию о том, как взаимодействует клиент и сервер и провести анализ на внедрение своих запросов и выявить уязвимости. Например для тестирования сайтов на предмет слабого механизма фильтрации входных параметров.





7.3. Selenium

Selenium - это инструмент для автоматизированного управления браузерами.

Наиболее популярной областью применения Selenium является автоматизация тестирования веб-приложений. Однако с его помощью можно автоматизировать любые другие рутинные действия, выполняемые через браузер.

Разработка Selenium поддерживается производителями популярных браузеров. Они адаптируют браузеры для более тесной интеграции с ним, а иногда даже реализуют встроенную поддержку Selenium в браузере. Он является центральным компонентом целого ряда других инструментов и фреймворков автоматизации. Поддерживает десктопные и мобильные браузеры. Selenium позволяет разрабатывать сценарии автоматизации практически на любом языке программирования. Selenium – это проект, в рамках которого разрабатывается серия программных продуктов с открытым исходным кодом (open source).

Какая часть Selenium нужна Вам?

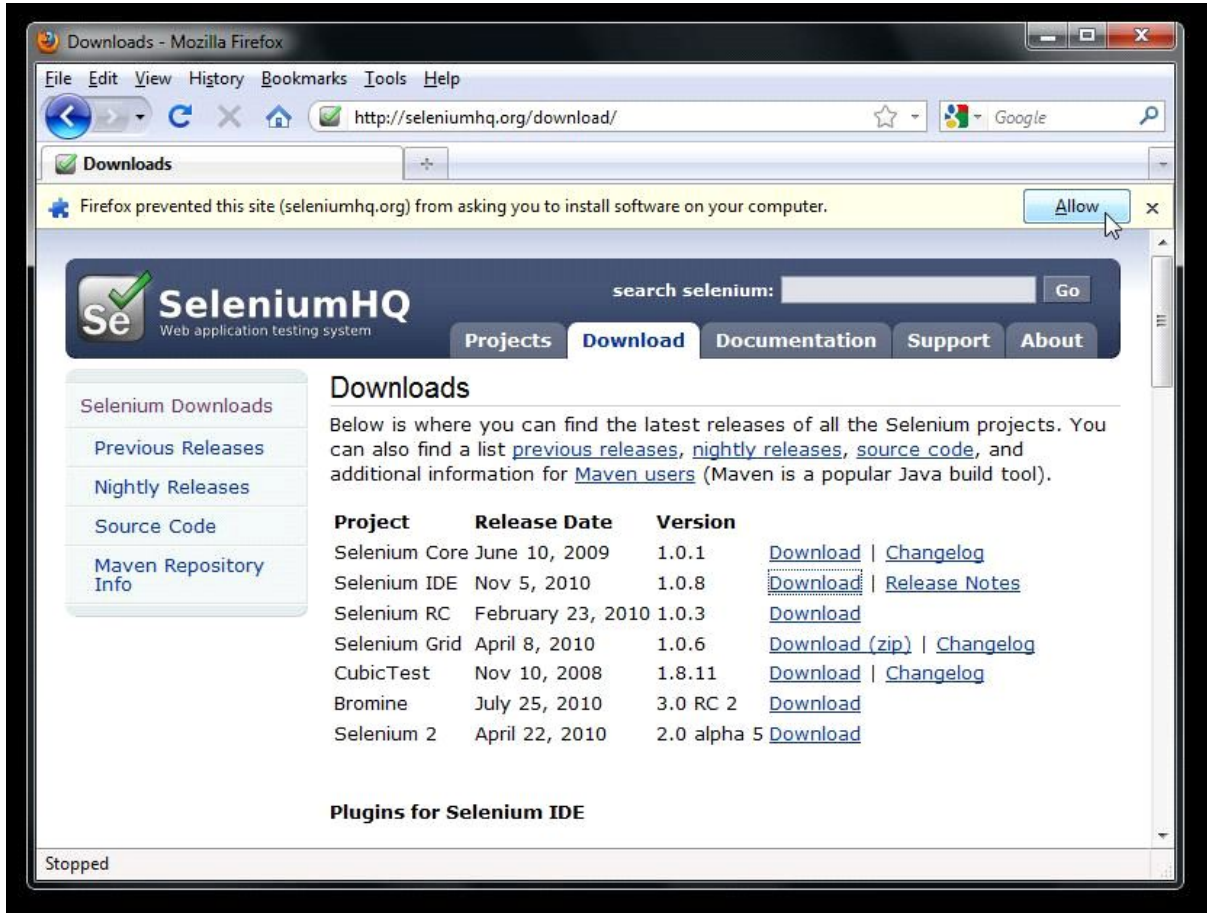
	<p>Если Вы хотите сделать:</p> <ul style="list-style-type: none"> ■ небольшой сценарий для быстрого автоматизированного воспроизведения бага, ■ вспомогательный скрипт для выполнения отдельных рутинных действий при ручном тестировании, <p>Вам нужен Selenium IDE – расширение браузера Firefox, которое позволяет записывать и воспроизводить действия пользователя в браузере.</p>
	<p>Если Вам требуется разработать:</p> <ul style="list-style-type: none"> ■ надежный фреймворк автоматизации, способный работать с любым браузером, ■ большой тестовый набор, включающий тесты с достаточно сложной логикой поведения и проверок, <p>Вам нужен Selenium WebDriver – набор библиотек для различных языков программирования, позволяющих управлять браузером из программы, написанной на этом языке программирования.</p>
	<p>Предшественником Selenium WebDriver является инструмент Selenium RC, который в настоящее время имеет статус "замороженного" и в дальнейшем развиваться не будет.</p> <p>Поэтому, если Вы ещё продолжаете использовать Selenium RC, рекомендуем Вам рассмотреть варианты миграции на Selenium WebDriver.</p>
	<p>Если Вам необходимо:</p> <ul style="list-style-type: none"> ■ запускать тесты удалённо на разных машинах с разными операционными системами и браузерами, ■ организовать тестовый стенд для выполнения большого количества тестов, <p>Вам нужен Selenium Server – он может принимать команды с удалённой машины, где работает сценарий автоматизации, и исполнять их в браузере. Несколько серверов Selenium могут образовывать распределённую сеть, которая называется Selenium Grid, что позволяет легко масштабировать стенд автоматизации.</p>

Selenium IDE (Integrated Development Environment - встроенная среда разработки): плагин к браузеру Firefox, который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия. В общем, это «Selenium-рекордер».

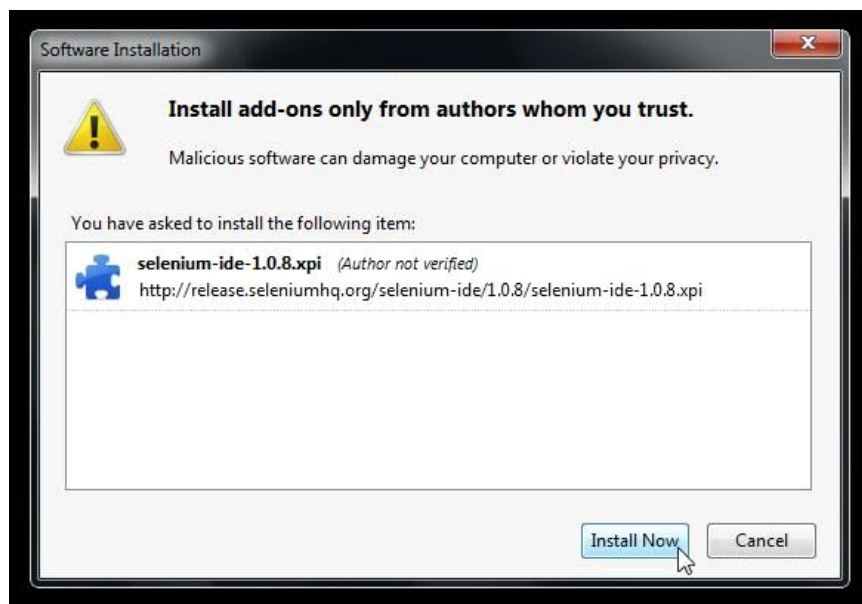
Selenium IDE редко,но все же используется как самостоятельный продукт, без преобразования записанных сценариев в программный код. Это, конечно, не позволяет разрабатывать достаточно сложные тестовые наборы, но иногда хватает и простых линейных сценариев.

Установка Selenium IDE

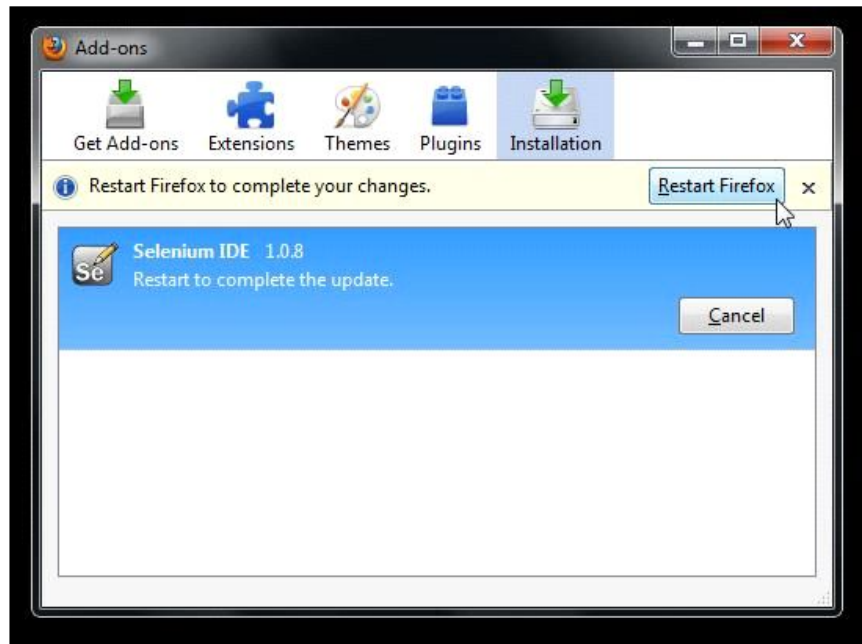
Запускаем Firefox, заходим на сайт <http://seleniumhq.org/download/> и далее все по инструкции:



Как только файл загрузится, нажимаем «Установить» и ждем полной установки плагина:

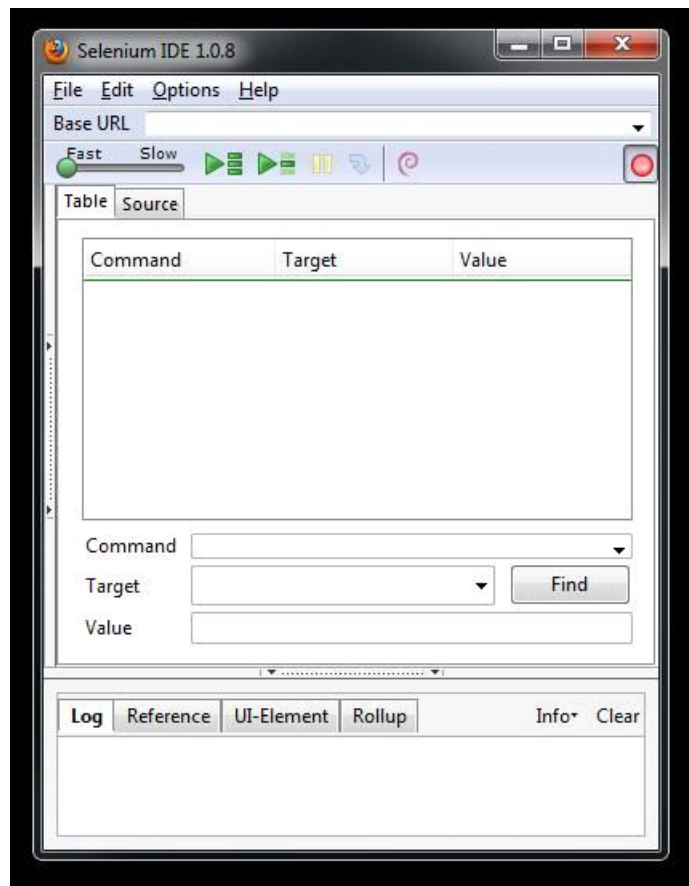


После завершения установки перезагружаем браузер и на этом процесс установки заканчивается:



Чтобы запустить Selenium IDE, просто выберите его из меню “Инструменты” браузера Firefox

Дополнение откроет пустое окно, предназначенное для редактирования тестовых сценариев, а также меню для их загрузки или сохранения.



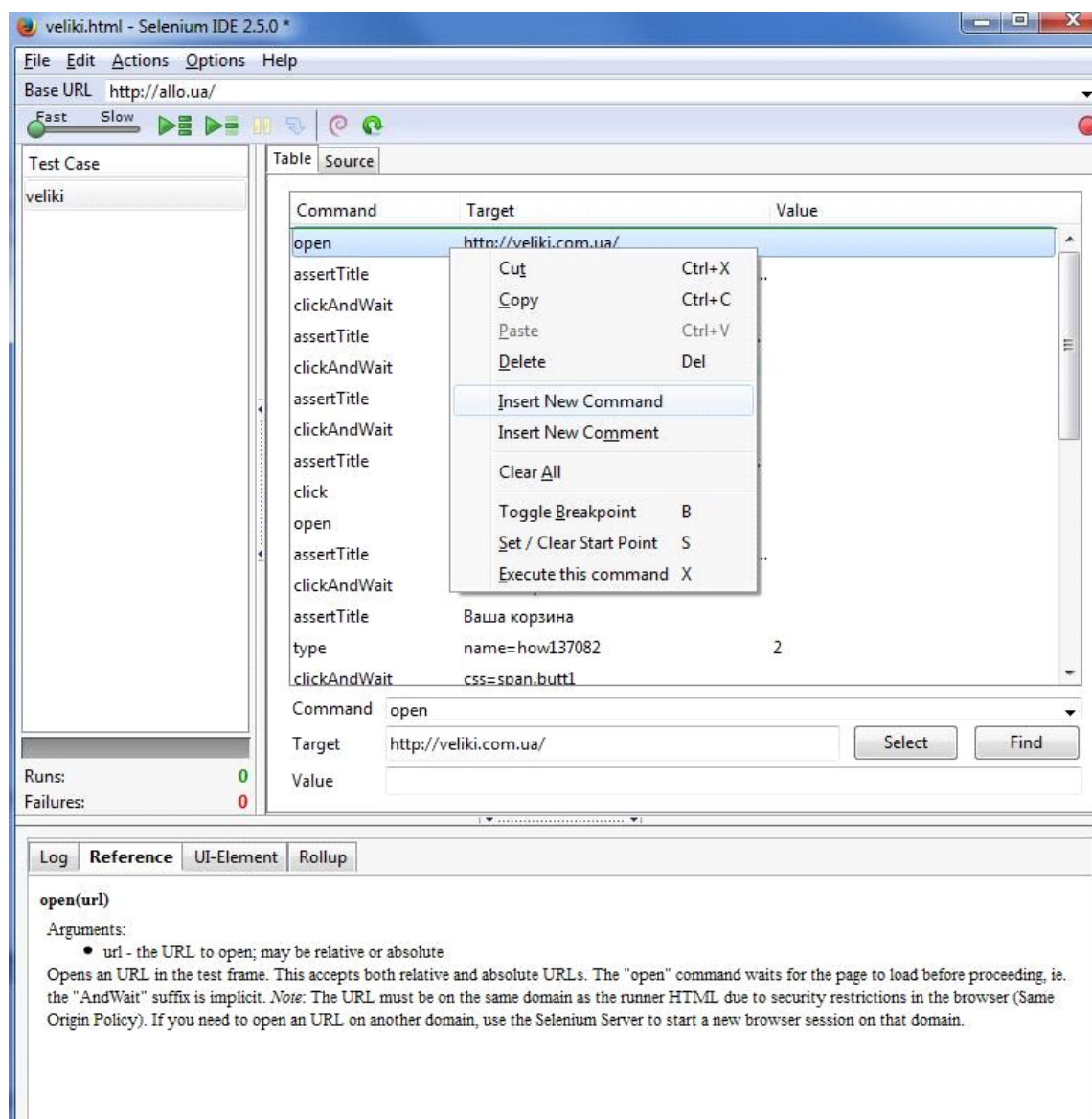
Начнем автоматизировать тест. Есть несколько вариантов:

- Активировать запись теста. В данном случае все пользовательские действия в браузере будут записаны, затем их можно будет отредактировать. Для старта теста нужно нажать на кнопку “Record”;
- Писать тест “руками”, т.е. заносить все команды самостоятельно. .

Естественно, более удобно использовать первый вариант записи тестов – быстро записать основные действия, а затем только отредактировать тест, добавив проверки и дополнительные шаги.

IDE предоставляет два вида отображения написанных команд: Table и Source. Удобнее разрабатывать тесты в виде Table, поэтому активируем данный вид отображения, нажав на вкладку Table, которая находится под зелеными кнопками запуска теста;

Пробуем добавить новую команду. Нужно выбрать в контекстном меню команду Insert new command;



Обратите внимание на следующие функции:

- В поле Command работает фильтр по названию команды, т.е. можно вводить имя команды, и вам будут выводиться команды начинающиеся на введенное значение. Это значительно ускоряет поиск команд.
- Справа от поля Target есть кнопка Find – с помощью данной кнопки можно проверять локаторы при написании тестов (находит ли Selenium указанный локатор). При нажатии на нее, если элемент на странице найден, – он будет выделяться желтым цветом, если не найден – то в логе увидите ошибку. Помните, что данная кнопка ищет только локаторы.
- Для того, чтобы узнать работает ли команда, не обязательно запускать тест полностью, достаточно дважды нажать на имя команды в таблице – выполнится только указанная команда.
- При написании тестов можно использовать команды из контекстного меню браузера на проверяемой странице, что значительно ускоряет процесс

создания тестов. Например, вы можете выбрать любой элемент страницы, вызвать для него контекстное меню и выбрать доступную команду для данного элемента. Данное действие запишется в тест и затем будет доступно для редактирования.

- В Log хранятся данные о запуске теста (логирование).
- В Reference предоставляется справочная информация о вызываемой команде.
- Можно устанавливать для команд breakpoint - точку останова. Смотреть в контекстном меню IDE для выбранной команды.
- Примечание 8. При написании тестов не забываем вставлять комментарии. Для этого в IDE вызываем контекстное меню и выбираем Insert New Comment. Затем в поле 'Command' вносим комментарий к тесту.

Команды Selenium

В Selenium существует три типа команд:

- Действия – функциональное действие над тестируемым веб-приложением в браузере. Например, заполнение полей, нажатие на кнопку и другие;
- Проверки – выполнение проверок на тестируемой странице. Например, проверка того, что определенное поле формы имеет указанное значение, или проверка заголовка окна;
- Ожидания – организация как, сколько и какое событие Selenium будет дожидаться (ожидания загрузки страницы, ajax и т.д.).

Действия (actions)

Набор действий в Selenium достаточно широк, основные действия:

- **open** – открыть страницу в браузере по определенному адресу.

Синтаксис команды – Open(string url).

Пример использования - selenium.Open("<http://blogs.logicsoftware.net/qa/>");

- **click** – произвести нажатие по элементу страницы.

Синтаксис команды – Click(string locator).

Пример использования – selenium.Click("LoginButton").

- **type** – ввести значение в текстовое поле страницы.

Синтаксис команды – Type(string locator, string value).

Пример использования - selenium.Type("id_TextField_1", "test");

- **select** – выбрать значение из выпадающего списка.

Синтаксис команды – Select(string selectLocator, string optionLocator).

Пример использования – selenium.select (TimeEntryTaskList, "Activity1") . В качестве опции для выбора элемента можно использовать следующие локаторы: label, value, id,

index. При использовании локатора label для optionLocator, можно искать элемент по частичному совпадению label=regexp:Locator;

- **selectWindow** – переключить фокус на другое окно.

Синтаксис команды – `SelectWindow(string windowID)`.

Пример использования – `selenium.selectWindow("id_dashboard");`

- **getTitle** – возвращает Title для текущей страницы.

Синтаксис команды – `GetTitle()`.

Пример использования – `selenium.GetTitle();`

- **getValue** – возвращает значение элемента страницы.

Синтаксис команды – `GetValue(string locator)`.

Пример использования – `selenium.GetValue("id_TextBox1");`

- **goBack** – вернуться на предыдущую страницу.

Синтаксис команды – `GoBack()`.

Пример использования – `selenium.GoBack();`

- **close** – закрыть текущее окно.

Синтаксис команды – `Close()`.

Пример использования – `selenium.Close();`

Локаторы

При использовании команд Selenium, сперва стоит научиться работать с локаторами.

Локатор (locator) – это строка, уникально идентифицирующая элемент веб-страницы. То есть то, с помощью чего ищутся элементы на странице.

Для того, чтобы выполнить действие на странице Selenium, необходимо знать, с каким элементом на странице ему нужно выполнить требуемое действие. Для этого и служат локаторы. Практически для всех команд первым параметром идет именно локатор, поэтому стоит сразу учиться правильно подбирать локаторы.

Типы локаторов в Selenium:

- **id** – в качестве локатора используется атрибут id (уникальный идентификатор) элемента страницы;
- **name** – в качестве локатора используется атрибут name элемента страницы;
- **identifier** – используется атрибут id элемента, если по id элемент не найден то поиск будет вестись по атрибуту name;
- **dom** – поиск элемента происходит по DOM выражению;
- **xpath** – используется для поиска элемента по XPath выражению;
- **link** – поиск ссылок с указанным текстом;
- **css** – данный тип локаторов основан на описаниях таблиц стилей (CSS).

Проверки (checks)

Проверки одна из главных составляющих при написании теста. В Selenium IDE выделяется два типа проверок – **assert** и **verify**. Различие между ними заключается в том, что если тест “падет” при выполнении проверки **assert**, то тест прекращает свою работу и помечается как failed. А если не выполняется проверка **verify**, то тест отметит данную проверку как failed, но продолжит свою работу.

В Selenium RC имена команд-проверок немного отличаются, нету деления проверок на **assert** и **verify**, вызов проверки начинается с **is**.

Например, команда в Selenium RC – `isElementPresent(string locator)`, в Selenium IDE данная команда будет иметь следующий вид `verifyElementPresent (locator)`.

Набор проверок в Selenium также достаточно широк. Команды, представленные для Selenium IDE:

- `verifyLocation / assertLocation` – проверить адрес текущей страницы.
Синтаксис команды – `verifyLocation(URL);`
- `verifyTitle / assertTitle` – проверить значение Title страницы.
Синтаксис команды – `verifyTitle (Title);`
- `verifyValue / assertValue` – проверить значение элемента страницы.
Синтаксис команды – `verifyValue (locator, value);`
- `verifyTextPresent / assertTextPresent` – проверить, что страница содержит указанный в команде текст.
Синтаксис команды – `verifyTextPresent (value);`
- `verifyElementPresent / assertElementPresent` – проверить, есть ли на странице указанный элемент.
Синтаксис команды – `verifyElementPresent (locator)`.

Ожидания (wait)

При выполнении некоторых команд, необходимо дожидаться загрузки страницы, или ее определенных элементов. Для этого предназначены команды-ожидания.

- `WaitForCondition` – ожидание выполнения определенного события на странице, указанного в параметре `script` (например, загрузка определенного элемента страницы, или страницы в целом).
Синтаксис команды – `WaitForCondition(string script, string timeout);`
- `WaitForFrameToLoad` – ждет загрузки фрейма на странице указанное количество времени.
Синтаксис команды – `WaitForFrameToLoad(string frameAddress, string timeout);`
- `WaitForPageToLoad` – ждет загрузки страницы указанное количество времени.
Синтаксис команды – `WaitForPageToLoad(string timeout);`

- WaitForPopUp – ждет появления PopUp элемента страницы указанное количество времени.

Синтаксис команды – WaitForPopUp(string windowID, string timeout).

Дополнения

Нибулон

Задание: используя разбиение на классы эквивалентности и анализ граничных значений протестировать калькулятор скидок :

<http://www.nibulon.com/data/zakupivlya-silgospprodukcii/kalkulyator-znizhok.html>

Все необходимые требования к калькулятору описаны внизу страницы - какие значения для каких культур являются базисными, по какой формуле рассчитывается скидка. Определить, какие будут классы эквивалентности и граничные значения, и на основании этого протестировать калькулятор.

Дополнительно протестировать локализацию.

Найденные дефекты описать в Джире.

Hello world

Задание: написать чеклист, протестировать программу и оформить найденные дефекты в Джире.

Ссылка для скачивания <http://skillup.com.ua/files/server.zip>

Описание программы: Программа реализована как многопоточный сервер под Windows для обработки запросов клиентов по TCP/IP. Параметры запуска:

helloworld.exe [port]

Функциональные требования:

- По умолчанию сервер слушает на порту 4010.
- [port] - квадратные скобки говорят о том, что параметр не обязателен. Если параметр не указать, сервер будет слушать по умолчанию порт 4010. Если указать любой другой - будет слушать указанный (если он не занят).
- При подключении клиента (любая программа) на слушающий порт, сервер пишет в лог, что связь с клиентом установлена.

- Если клиент отправляет запрос (один байт - любое значение), после чего программа сразу же присылает в ответ тест «Hello, World» и закрывает соединение (пишет об этом в лог).
- Программа принимает до 5 одновременных подключений клиентов.
- Таймаут ожидания запроса клиента на после установки соединения – 20 сек, после истечения таймаута программа закрывает соединение

Нефункциональные требования:

- Высокая скорость обработки запросов клиентов.
- Надежность, стабильность работы.

Задача: Обнаружить ошибки в программе Hello World 2000

Проверять работу приложения Helloworld будем с помощью программы telnet. Telnet - сетевой протокол для реализации текстового интерфейса по сети.

Как подключиться к серверу и отправить данные:

1. Включить telnet в Windows: Пуск → Панель управления → Программы и компоненты → Включение или отключение компонентов windows → Клиент Telnet (ставим галочку) → ok.
2. Для запуска: открыть командную строку: Win+R → cmd → OK
3. В командной строке ввести **telnet localhost 4010**. Эта команда откроет соединение между клиентом и сервером, который находится на том же компьютере что и клиент (localhost), на порту 4010. Если же сервер будет находиться на другой машине, вместо localhost необходимо будет ввести IP-адрес сервера.
4. После этого нужно проверить лог сервера. Там вы увидите, что клиент 1 подключился.

5. После этого ввести любой символ в окне клиента - т.е. отправить информацию на сервер.
6. Сервер отвечает "Hello World!" и закрывает соединение. Соответственно, клиент выйдет из программы telnet обратно в командную строку.

Для того, чтобы посмотреть какие соединения (сокеты) открыты, использовать команду:

```
>> netstat -an
```

Полезные ссылки

Описание	Ссылка
Программа QALight	https://drive.google.com/drive/u/1/folders/0B0AEkvOqt8KYcklFTTNqaXdxRU0
Курс Тестирование ПО	https://www.udemy.com/qa-software-testing-training-course/
О тестировании	http://software-testing.ru/
Походная книга тестировщика (книги на русском)	https://habrahabr.ru/post/184962/
Школа начинающих тестировщиков	http://testbase.ru/
Помощь тестировщику	http://qahelp.net/
Тестирование. Фундаментальная теория	http://dou.ua/forums/topic/13389/
Pairwise testing	http://forworktests.blogspot.com/2013/11/pairwise-testing.html
Network, client-server	https://habrahabr.ru/company/htmlacademy/blog/254825/
CLI	https://technet.microsoft.com/en-us/library/bb490890.aspx
Юнит тесты	http://rsdn.org/article/testing/UnitTesting.xml
HTTP стандарт	https://tools.ietf.org/html/rfc2616
Онлайн курсы	https://www.edx.org/courses

Онлайн курсы	https://www.coursera.org/
Украинский ресурс	http://prometheus.org.ua/
Изучение Java	http://javarush.ru/

Рекомендации на время курса и далее

За время курса желательно прочесть следующие книги:

- 1) С. Куликов “Тестирование программного обеспечения. Базовый курс”
http://svyatoslav.biz/software_testing_book/
- 2) С. Канер “Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений”
http://aniri.flatrate.ru/Reading/Kompjuternaja%20literatura/kaner_testing.pdf
- 3) Р. Калбертсон “Быстрое тестирование”
http://adm-lib.ru/books/2/bstroie_testirovanie.pdf
- 4) Дж. Уиттакер “Как тестируют в Google”
<http://www.rulit.me/books/kak-testiruyut-v-google-read-365579-1.html>

Также, желательно, найти подработку на любом открытом проекте, где вы сможете видеть процесс разработки, и заводить найденные ошибки. Позже вы сможете этот опыт внести в свое резюме, что существенно повысит ваши шансы найти работу

Дополнительно почитать:

- 1) Э. Таненбаум “Современные операционные системы”
- 2) Э. Таненбаум “Компьютерные сети”

Мир тестирования и разработки полон противоречий. Для избежания споров по многим вопросам существует стандартизация и сертификация. Одна из самых авторитетных из них **ISTQB**. Если у вас возникают противоречия или сомнения по материалам, рекомендуется посещать глоссарий:

<http://www.rstqb.org/sertifikacija/materialy/materialy-istqb.html>

Данные сертификаты все более востребованы на рынке как Украины так и европейских стран.

Вы можете пройти курс подготовки к сдаче сертификата в учебном центре SkillUp:

<http://skillup.ua/278-istqb-certification>

Для более глубокого изучения автоматизации, и освоения профессии QA automation необходимо знание основ программирования и специальных инструментов. Для этого существуют специальные курсы с практикующими автоматизаторами:

<http://it.skillup.com.ua/kyrsu-avtomatizacii/>